# EE323 Project 4

# STCP: Implementing a Reliable Transport Layer

June 1, 2020

TA: Soobin Lee, Mangi Cho

EE323TA@gmail.com

# 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 and server program work in a **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 **KLMS PDF file** and **RFC 793** carefully

- Download the STCP tarball from the KLMS and extract it on one of the lab machines
  - $ tar xzvf assignment4.tar.gz

- Check any compile errors with current Makefile
  - It should compile and run on any lab machines
  - The server and client will compile, but NOT run: They are dummy

# Code Structure

Application Layer

- Simple, dummy client / server
  1. Client sends a request for a file
  2. Server transmits a file to client
  3. Client saves the transmitted file locally with filename ‘recv’
- Help with debugging of your transport layer

**Transport Layer: your task!**

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

Network Layer

- Emulates an unreliable datagram communication service
- Reliable or ~~unreliable~~ modes with client / server option -U
  - Default : reliable mode
  - ~~-U : unreliable mode~~
- Interfaces for transport layer is defined in stcp_api.h

mysock.h

stcp_api.h

# MYSOCK Layer Overview

**Warning: this ppt does not cover the complete specification.**
**Please read the assignment material carefully.**

- Important implementation items
  - 3-way handshaking for connection establishment
  - Sequence number semantics for packets
  - Sliding windows for receiver and sender windows
    Please check the rules for managing the windows
  - Slow start
  - 4-way handshaking for connection teardown
  - NO TCP option handling
  - No optimization for small data:
    Send data packets as soon as data is available from the application
  - No delayed ACK:
    Send ACK packets as soon as data is received

# 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

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

- A connection is initialized by calling transport_init();
- transport_init(); should not return until the connection ends

# Implementation Hints

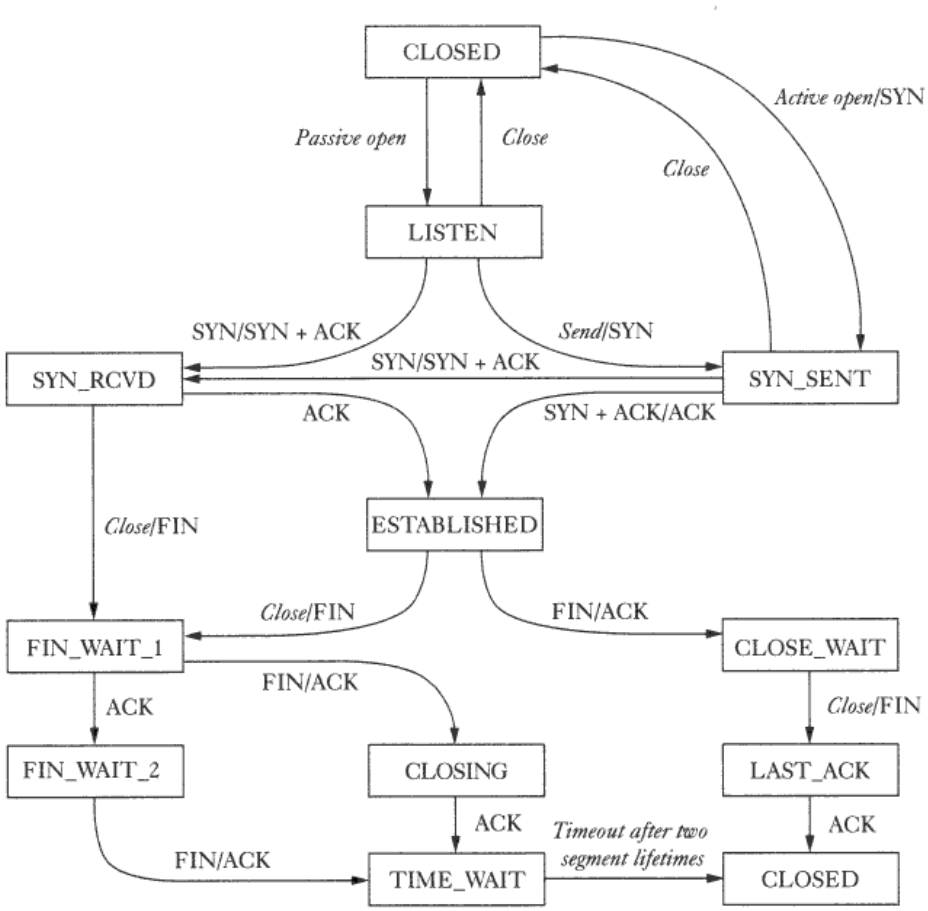
- Implement the TCP state diagram
- Need to have your **TCP context** (struct context_t) maintain a current state and react to the events from the application or the network layer possibly to transition from one state to other
- Include **header files** and **define** constants if you need
  - SEND_WINDOW_SIZE , RECV_WINDOW_SIZE, CWND_SIZE, PAYLOAD_SIZE, …,
- Add **STCP states** at enum
  - CSTATE_CLOSED, CSTATE_LISTEN, CSTATE_SYN_SENT, CSTATE_SYN_RCVD, CSTATE_ESTABLISHED, CSTATE_FIN_WAIT_1, CSTATE_FIN_WAIT_2, CSTATE_CLOSE_WAIT, CSTATE_LAST_ACK, CSTATE_CLOSING
- Add **structures** if you need
  - STCPPacket, RxSegment, TxSegment, …
- Create **functions** as freely as you want

# TCP State Diagram

- Clearly understand the TCP state diagram by reading RFC 793

- What does each state mean?
- How transitions can be happened?
- What should be done in the current state?

- Check the differences between real TCP and our STCP



# transport_init()

- All your STCP processing starts from here
- 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())
  - Create and send a **SYN segment** and mark your state to **SYN_SENT**
  - You may need to manage Tx packets to check ACK
- If **is_active == FALSE**, your application is listening on a port
  - Your TCP state should be **LISTEN**
- Call **control_loop()** for the main process

# control_loop()

- **Main processes** are described in this function
- Get an **event** using **stcp_wait_for_event()**
  - Incoming data from the peer
  - New data from the application via mywrite()
  - The socket to be closed via myclose()

- Do appropriate jobs considering the event
  - Check the state, change the state, and send a packet, etc.
  - Close the connection
- Use **TCP context** to store the state and other necessary information
- Exit the control-loop when the connection is finished
- Use functions in **stcp_api.c** and **stcp_api.h**

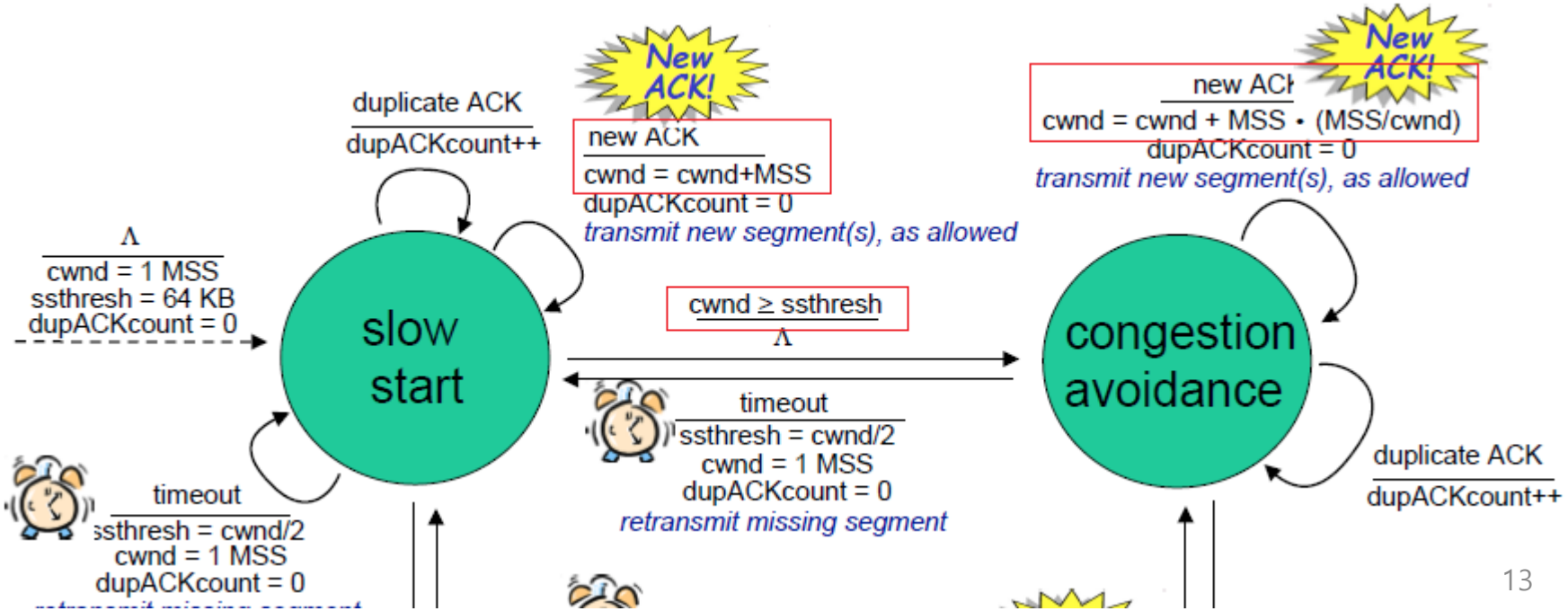
# stcp_api.c & stcp_api.h

- Network layer interface for transport layer is defined in stcp_api.h
- Important functions
  - stcp_unblock_application()
  - stcp_wait_for_event()
  - stcp_network_recv()
  - stcp_network_send()
  - stcp_app_recv()
  - stcp_app_send()
  - stcp_fin_received()
- See the descriptions on stcp_api.h for more details
- Highly recommend to study the implementation of these functions in network layer

# Sliding window & Slow start

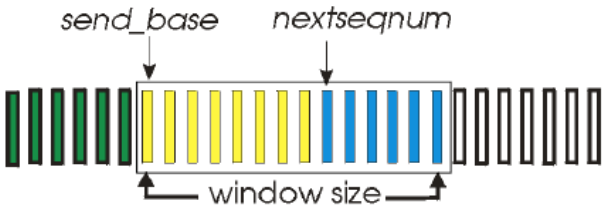
- You have to manage two window, cwnd, rwnd.
  - swnd = Min(cwnd, rwnd)

- cwnd increases with rule of slow start
  - Begin with 1MSS and threshold is 4 MSS



# Print logs

- Whenever you send data or whenever you receive ACK, print log of (Use fopen, fprintf):
  - swnd : full sender window size
  - Rem : window size that usable but not yet sent in sender window
    - Swnd – (NextSeqNum – SendBase)
  - Byte : size of data which is sent or is ACKed.




- Format :
  - Make 2 log files "client_log.txt", "server_log.txt"
  - "Send:₩t*swnd*₩t*Rem*₩t*Byte*₩n"
    - Whenever you send data (not only ACK)-> stcp_network_send()
  - "Recv:₩t*swnd* ₩t*Rem*₩t*Byte*₩n"
    - Whenever you receive ACK -> stcp_network_recv(), th_flags & TH_ACK

# Print logs (Hint)

- It can be confusing when you should print log exactly.
- Part of code structure might be :

```
If ( event & APP_DATA) {                    /* there is something to send*/
        if meet conditions for sending      /* window condition */
                stcp_network_send();        /* send packet */
                **print sending log here**
                post-processing for sliding window …
}

If ( event & NETWORK_DATA) {          /* there is something to receive*/
        stcp_network_recv();          /* receive packet */
        if the packet is ACK
                **print receiving log here**
                post-processing for sliding window …
}
```

# Testing Your STCP

- Run given **client** and **server** on different shells (2 PuTTY)
  - $ ./server
    - Check the **port number** that is bound by server randomly
  - $ ./client [SERVER_IP_ADDR]:[SERVER_PORT_NUM]
- Give **the name of the file** to client
  - You can use *–f* [FILENAME] option to give the name of the file to be transferred
    - $ ./client [SERVER_IP_ADDR]:[SERVER_PORT_NUM] –f [filename]
  - 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**"
- Use **diff** command to compare the original file and "rcvd"

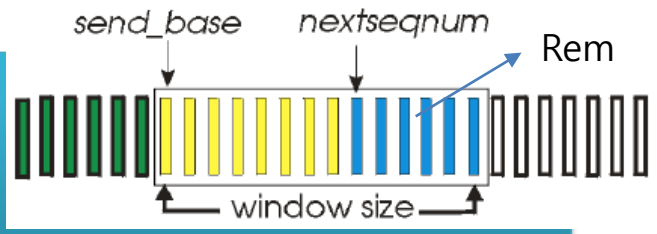
# Testing Your STCP

- When you run client run-time (not use –f option), client and server don't call myclose() until forced quit.

- So, we will check log files only with –f option.
- Solution file also print logs properly only when –f option.

- We will provide additional code for check logics of log files.

# Demo (server)

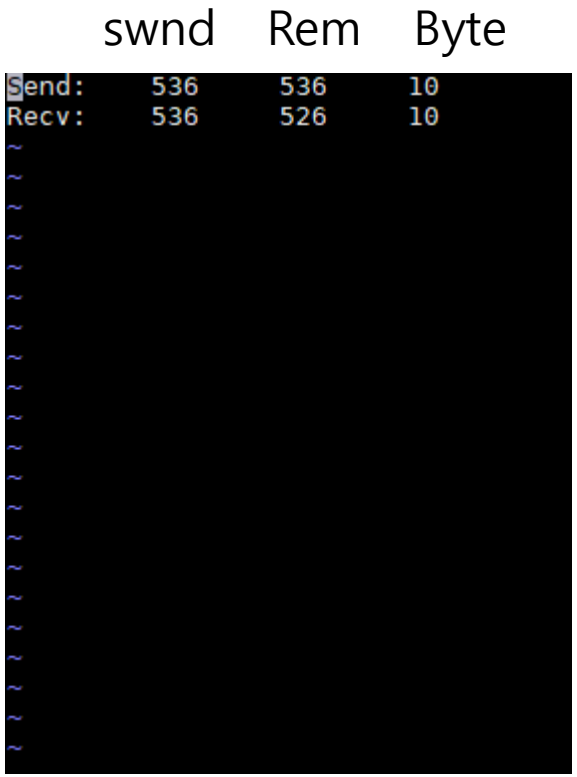



| | swnd | Rem | Byte |
|---|---|---|---|
| Send: | 536 | 536 | 19 |
| Send: | 536 | 517 | 517 |
| Recv: | 536 | 0 | 19 |
| Send: | 1072 | 555 | 536 |
| Recv: | 1072 | 19 | 517 |
| Send: | 1608 | 1072 | 536 |
| Send: | 1608 | 536 | 536 |
| Recv: | 1608 | 0 | 536 |
| Send: | 2144 | 1072 | 536 |
| Recv: | 2144 | 536 | 536 |
| Send: | 2278 | 1206 | 536 |
| Recv: | 2278 | 670 | 536 |
| Send: | 2404 | 1332 | 536 |
| Send: | 2404 | 796 | 536 |
| Send: | 2404 | 260 | 260 |
| Recv: | 2404 | 0 | 536 |
| Send: | 2523 | 655 | 471 |
| Recv: | 2523 | 184 | 536 |
| Send: | 2636 | 833 | 536 |
| Recv: | 2636 | 297 | 536 |
| Send: | 2744 | 941 | 536 |
| Recv: | 2744 | 405 | 536 |
| Send: | 2848 | 1045 | 536 |
| Recv: | 2848 | 509 | 260 |
| Send: | 2948 | 869 | 536 |
| Send: | 2948 | 333 | 333 |
| Recv: | 2948 | 0 | 471 |
| Send: | 3045 | 568 | 536 |
| Recv: | 3045 | 32 | 536 |
| Send: | 3072 | 595 | 536 |
| Recv: | 3072 | 59 | 536 |
| Send: | 3072 | 595 | 536 |

- Initially, swnd = 536, Rem =536.
- Client request file to server, then Server send ACK to client
- (1 line) As response to request, server send data to client. At that time, swnd=536, Rem = 536, and sending data size = 19
- (2 line) There are more data to send, but the Rem is 517 (=536-19) so only send 517 byte.
- (3 line) ACK for first packet received (ACKed size 19), at that time Rem is 0. Then swnd increase by MSS (slow start)
- (4 line) Therefore, Rem is 555 (=536+19) so we can send data with maximum size 536.

The order can be different!
We will only check logic

# Demo (client)




- Simple!!
- Send request data (10 byte)
- Then received ACK. At that time Rem is 526
- If you have question about these demos, post piazza.

# 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

# Cautions

- 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 one of the lab machines.
  - 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

- Submission
  - transport.c: works in reliable mode
  - README: one page
    - Describing the design of your transport layer
    - Any design decisions/tradeoffs that you had to consider
  - Zipped like “YourStudentID_assign4.tar.gz”
    - tar cvzf 20120000_assign4.tar.gz [source_files]
  - Due: 11:59 pm on 6/26 , at KLMS

# Others

- Do NOT copy and paste someone else'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 post piazza.