Part 1 Description:
We learn from the course that “encapsulation” is used frequently in networking. In this project, we will emulate TCP sender and receiver in application layer and call UDP protocol to transmit TCP packets. That is, TCP packets will be encapsulated and sent in UDP layer by the source and decapsulated in the receiver. **In this project, both directions need to be implemented.** Besides, there is an NAT translation at one side of the connection which you need to implement.
The TAs will tell you how to test your program. That is, TAs will write sources and/or receivers to test your program. Therefore, it becomes very important to follow TAs’ formats so your program can be test. The TCP segment structure is as enclosed and you must follow the segment format and implement at least the following fields: source port, destination port, sequence number, acknowledgement number, and checksum for primitive TCP, TCP Tahoe and TCP Reno. In this project, you are also asked to write a TCP flow control program with the Selective Acknowledgements (SACK) option. The state diagram is as shown in the last page of the project.

Step 1: First, you would write two sources (sender and receiver) which can both transmit packets and ACKs with the other. The round trip delay is first set to be 200 ms. The variable threshold is initially set to 65535 bytes. The MSS is 1500 bytes, 536, or 512 bytes. The TCP receiver is assumed to have a buffer size of 10KB packets.
Step 2: NAT is added.
Step 3: Delay ACKs are added to senders and receivers.
Step 4: The TCP congestion control consists of the slow start and the congestion avoidance mechanisms. In step 4, you implement the slow start mechanism and the congestion avoidance mechanism.
Step 5: The fast retransmit mechanism is added (Tahoe).
For the moment, there are still only two states, slow start state and congestion avoidance state. However, the fast retransmit mechanism is already included in TCP Tahoe. For TCP Tahoe, when a loss happens, fast retransmit is triggered and the slow start state is entered.
Step 6: The fast recovery mechanism is added (TCP Reno).
Step 7: Finally, the SACK option is incorporated into the standard.
Step 8: Channels may have losses.

The program consists of nine subprojects.

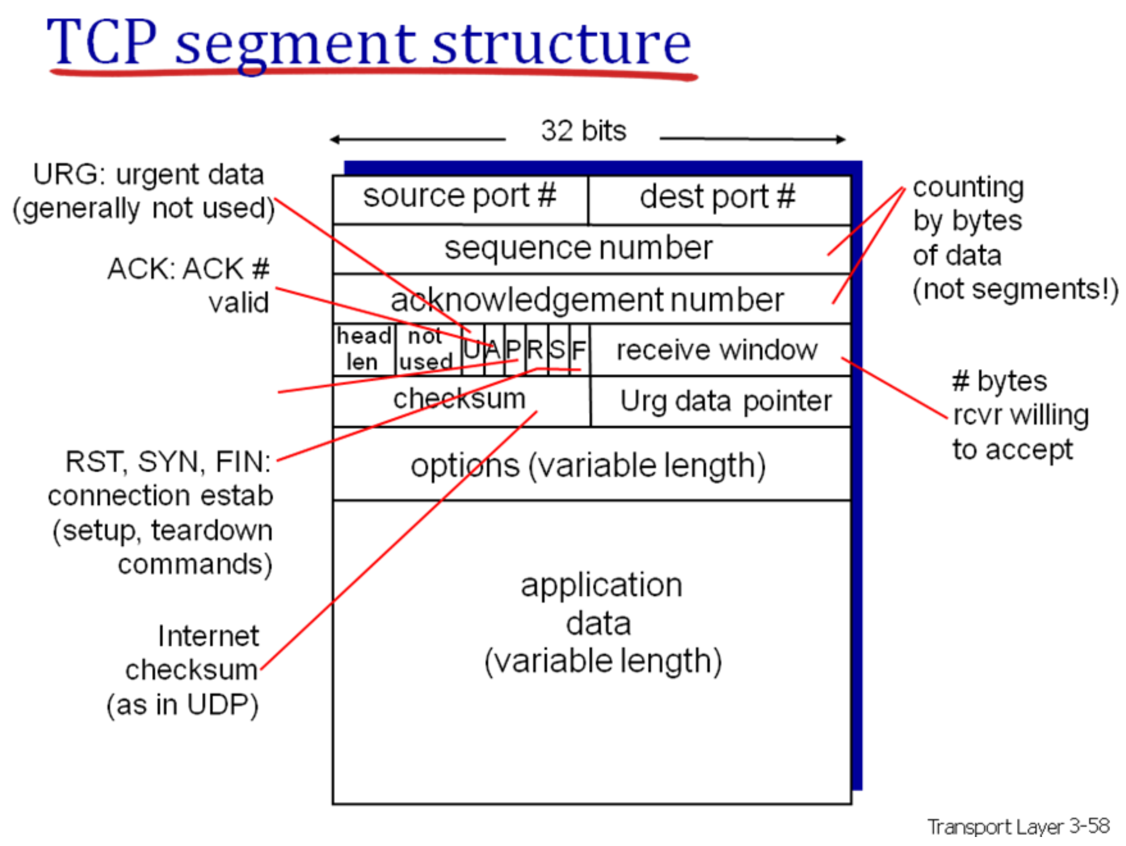
A. Both directions have the same transmission rate of 300 kbps.
(1) Sender and receiver. (10)
(2) NAT is implemented. Although there may have many TCP connections, there is only one TCP address available. Therefore, you need to build a mapping table and make the translation between (input IP, input port number) and (output IP, output port number). (10)

(3) Delay ACKs. (10)
(4) TCP slow start. (RTT = 200ms for both sides) and TCP congestion avoidance. (15)
(5) TCP with fast retransmit (TCP Tahoe). (10) (RTT = 200ms for both sides)
(6) TCP with fast retransmit and fast recovery (TCP Reno). (10) (RTT = 200 ms for both sides)
(7) The SACK option is added. (10)
B. There are loses.
(8) The link becomes unstable and there are 5% of loss rates. The loss rates are generated randomly. There are two connections in each direction. (RTT = 200ms for both sides). One direction has a transmission rate of 200 kbps and the other direction has a transmission rate of 50 kbps. Only TCP Reno is simulated for case B. (15)
(9) There are six connections in each direction. The loss rates are 2 % and generated randomly. One direction has a round trip delay of 500 ms and the other direction has a round trip delay of 200 ms. Transmission rates are 300 kbps. (10)

Due: June 20, 2017
References:
The first five papers are suggested for understanding the mechanisms. The last three papers are for additional readings. They are listed below. The last two references are about the relationship about bandwidth x delay and TCP receiver window size.

[1] F. M. Allman, V. Paxson, and W. Stevens, “TCP congestion control,” RFC 2581, Apr. 19999, http://www.rfc-editor.org/rfc/rfc2581.txt.
[2] S. Floyd and T. Henderson, “The New Reno Modification to TCP’s Fast Recovery Algorithm,” RFC 2582, Apr 1999, http://www.rfc-editor.org/rfc/rfc2582.txt.
[3] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “TCP selective acknowledge options, RFC 2018,” Oct. 1996, http://www.rfc-editor.org/rfc/rfc2018.txt.
[4] V. Jacobson and R, Braden, “TCP extensions for long-delay paths,” RFC 1072, http://www.rfc-editor.org/rfc/rfc1072.txt.
[5] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, six edition, Addison Wesley, Reading, 2012
[6] K. Fall and S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, Computer Communication Review, vol. 26, No. 3, July 1996.
[7] S. Floyd, “A report on some recent developments in TCP congestion control,” IEEE Communications Magazine, Apr 2001, http://www.aciri.org/floyd/papers/report_Jan01.pdf.
[8] W. R. Stevens, TCP/IP Illustrated vol. 1: The Protocols, Addison-Wesley, Reading, MA, 1994. (Due: Dec 17, 2007)
[9] http://speedtest.raketforskning.com/tuning-tcp-window-size.html
[10] http://en.wikipedia.org/wiki/TCP_tuning

PS. The TA will tell you how to test your program.



The SACK option is to be included in a segment sent from a TCP that is receiving data to the TCP that is sending that data; we will refer to these TCP's as the data receiver and the data sender, respectively. We will consider a particular simplex data flow; any data flowing in the reverse direction over the same connection can be treated independently.

2. Sack-Permitted Option

This two-byte option may be sent in a SYN by a TCP that has been extended to receive (and presumably process) the SACK option once the connection has opened. It MUST NOT be sent on non-SYN segments.

TCP Sack-Permitted Option:

Kind: 4

```
    +---------+---------+
    | Kind=4  | Length=2|
    +---------+---------+
```

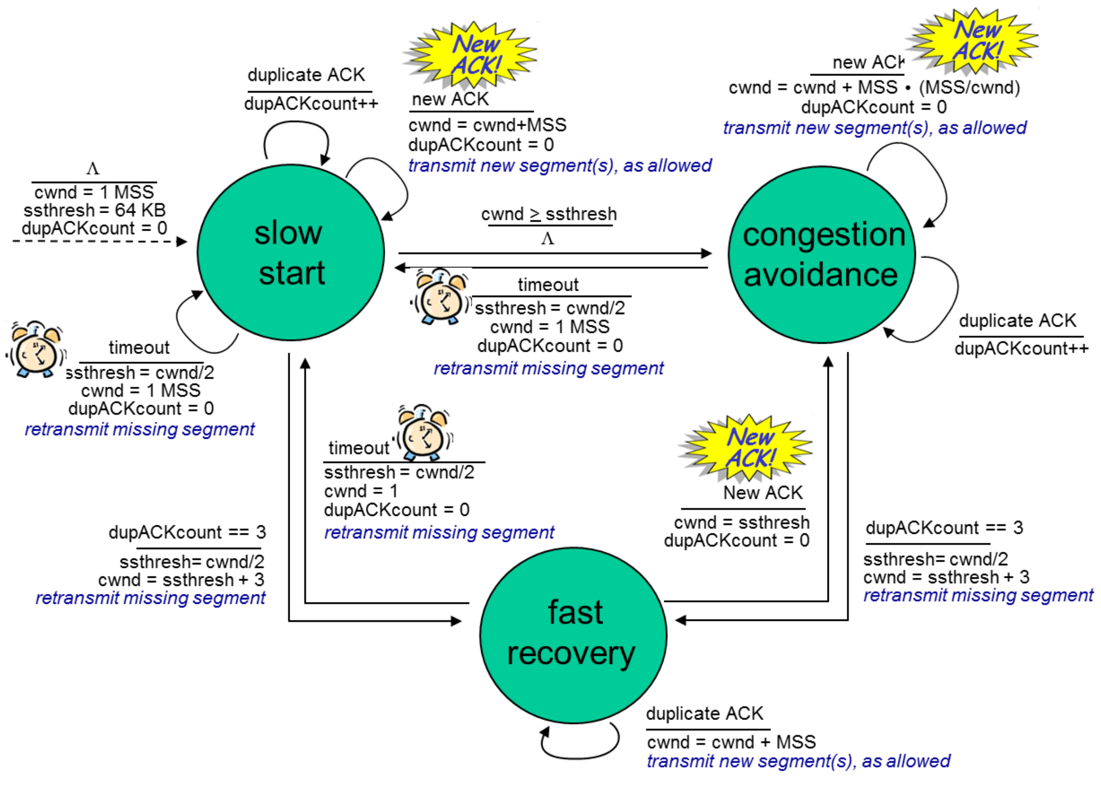
3. Sack Option Format

The SACK option is to be used to convey extended acknowledgment information from the receiver to the sender over an established TCP connection.

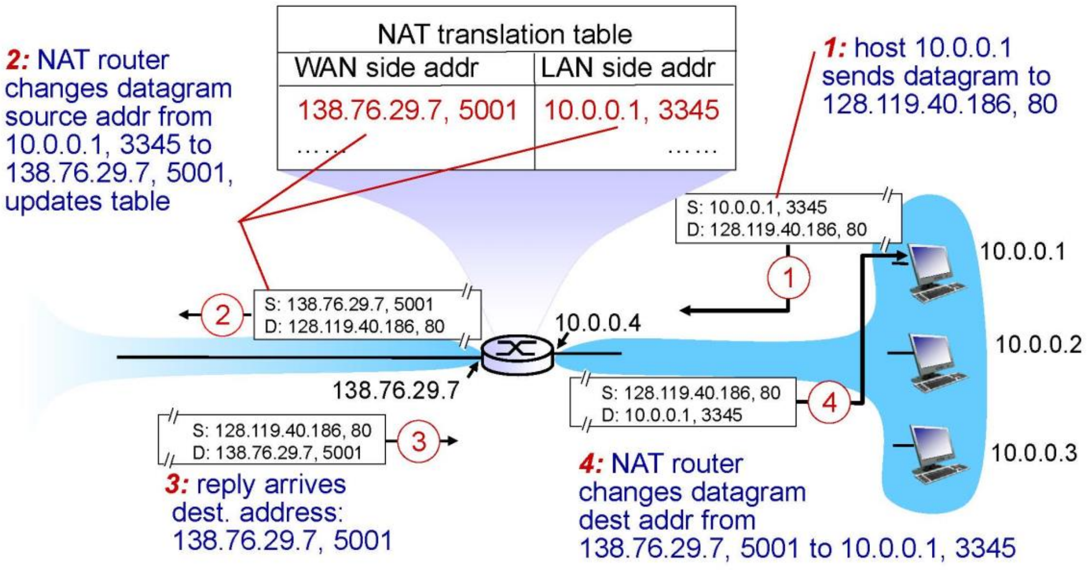
```
    TCP SACK Option:

    Kind: 5

    Length: Variable

                      +--------+--------+
                      | Kind=5 | Length |
    +--------+--------+--------+--------+
    |      Left Edge of 1st Block       |
    +--------+--------+--------+--------+
    |      Right Edge of 1st Block      |
    +--------+--------+--------+--------+
    |                                   |
    /            . . .                  /
    |                                   |
    +--------+--------+--------+--------+
    |      Left Edge of nth Block       |
    +--------+--------+--------+--------+
    |      Right Edge of nth Block      |
       +--------+--------+--------+--------+
```

Transport layer 3- 70, 3-104

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap



NAT: network address translation


NAT translation table
WAN side addr
LAN side addr
138.76.29.7, 5001
10.0.0.1, 3345
……
……
2: NAT router changes datagram source addr from 10.0.0.1, 3345 to 138.76.29.7, 5001, updates table
1: host 10.0.0.1 sends datagram to 128.119.40.186, 80
S: 10.0.0.1, 3345
D: 128.119.40.186, 80
1
10.0.0.1
10.0.0.2
10.0.0.3
2
S: 138.76.29.7, 5001
D: 128.119.40.186, 80
10.0.0.4
138.76.29.7
S: 128.119.40.186, 80
D: 10.0.0.1, 3345
4
S: 128.119.40.186, 80
D: 138.76.29.7, 5001
3
3: reply arrives dest. address: 138.76.29.7, 5001
4: NAT router changes datagram dest addr from 138.76.29.7, 5001 to 10.0.0.1, 3345