We learn from the course that “encapsulation” is used frequently in networking. In this project, we will emulate TCP sender and receiver in the application layer and call UDP protocol to transmit TCP packets. That is, TCP packets will be encapsulated and sent in the UDP layer by the source and decapsulated in the receiver. **In this project, both directions need to be implemented.**
The TAs will tell you how to test your program. That is, TAs will write clients to test your server and server to test your clients. Therefore, it becomes very important to follow TAs’ formats so your program can be tested. 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, acknowledgment number, and checksum for primitive TCP, TCP Tahoe, TCP Reno, and TCP SACK. The state diagram is as shown on the last page of the project. The server provides four video files for clients to retrieve. A client can request multiple video files within each request. The round trip delay is first set to be 15 ms for each connection. The variable threshold is initially set to 64 Kbytes. The MSS is 1 Kbytes. The TCP buffer size is assumed to have a buffer size of 512 Kbytes. The maximum transmission rate for each TCP connection from the server side to the client side is assumed to be two Mbps. The TAs will provide you with four video files whose sizes range between one Mbytes to three Mbytes.
Step 1: First, you would write one server and one client, and the server and client can both transmit packets and ACKs. The server must be able to handle multiple file requests from the client. The client would randomly request one to four video files at one time.
Step 2: The clients are increased to four clients.
Step 3: Channels may have losses. That is to say, both data and ACKs are subject to losses. You would begin to introduce losses for packets sent by the server or clients. For simplicity, we implement packet loss rates instead of channel lost rates. The packet error rate is set to have a Poisson distribution with a mean of 10^{-6}. For each generated packet, we first decide if it is lost. If we determine the packet to be a lost packet, we deliberately insert the wrong checksum in the checksum field of the packet.
Step 4: Delayed ACKs are added to senders and receivers.

Step 5: 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 6: 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. Only for this step and the next step, to trigger the fast retransmit you would gradually increase the packet loss rates until fast transmit is triggered.

Step 7: The fast recovery mechanism is added (TCP Reno). It would be a plus if both cases of returning to slow start state and returning to congestion avoidance state are demonstrated.

Step 8: The SACK option is incorporated into the standard.

Step 9: Now increase the number of clients to 30. The server side has to implement some prevention mechanisms to handle simultaneous requests from clients. One possible solution is suggested as follows. Since the parent process has to have a method to handle input requests, the parent process can fork some child processes responsible for various input requests from different clients.

The program consists of nine subprojects.

(1) A server and a client. (10)
(2) Four clients. (10)
(3) Losses added. (10)
(4) Delayed ACKs. (10)
(5) TCP slow start and TCP congestion avoidance. (20)
(6) TCP with fast retransmit (TCP Tahoe). (10)
(7) TCP with fast retransmit and fast recovery (TCP Reno). (10)
(8) The SACK option is added. (10)
(9) Increase the number of clients to 30. (10)

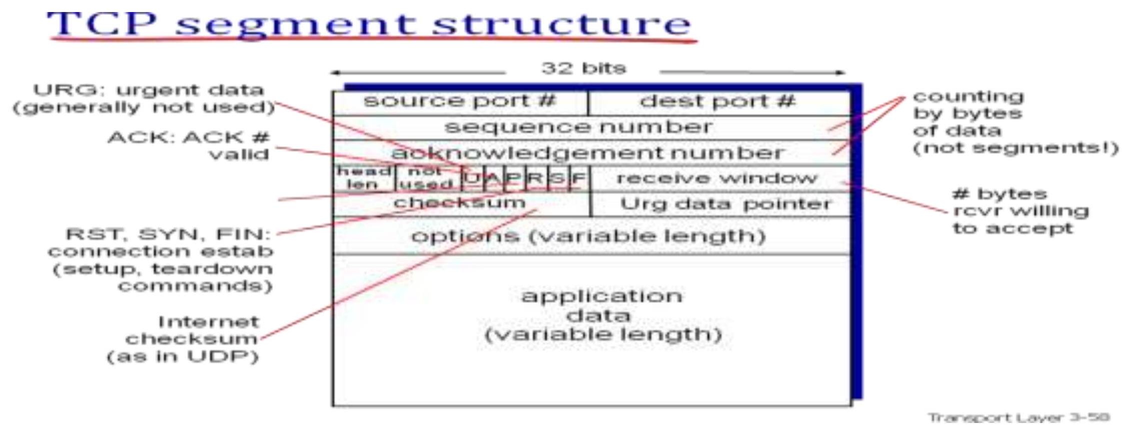
Due: June 18, 2020

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 between 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, sixth 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.


TCP segment structure
32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U A P R S F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
# bytes rcvr willing to accept
Transport Layer 3-58

RFC 2018   TCP Selective Acknowledgement Options   October 1996

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

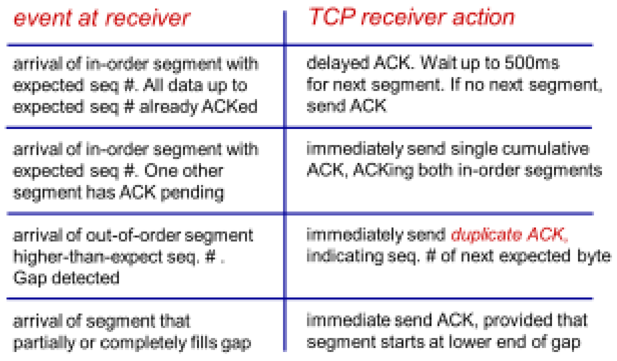
```
+---------+---------+
| Kind=4  | Length=2|
+---------+---------+
```

3. Sack Option Format

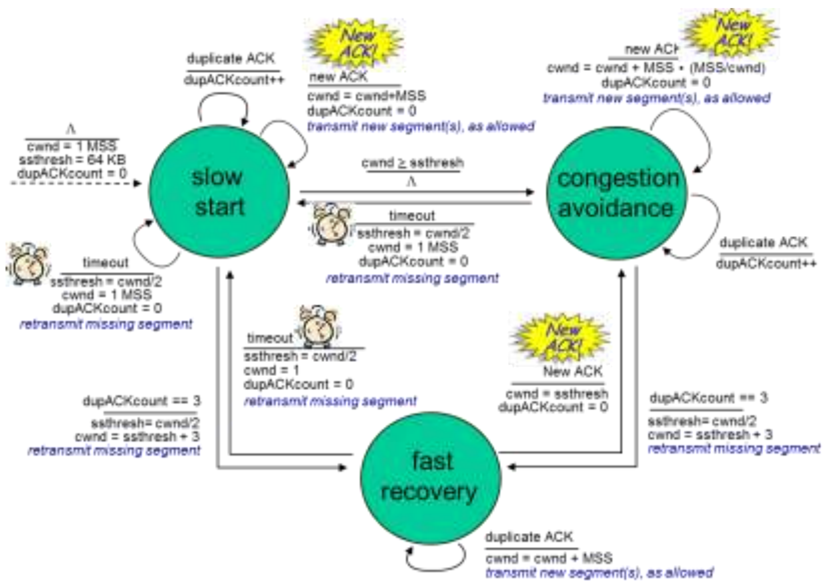
The SACK option is to be used to convey extended acknowledgment information from the receiver to the sender over an established TCP connection.

```
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



slow start
congestion avoidance
fast recovery
Λ
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0
duplicate ACK
dupACKcount++
New ACK!
new ACK
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed
New ACK!
new ACK
cwnd = cwnd + MSS • (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed
cwnd ≥ ssthresh
Λ
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
duplicate ACK
dupACKcount++
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
timeout
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment
New ACK!
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed