Please type your answers and submit the hardcopy output by 5pm in CS 164 on the due date. Submit the soft copies—programs, output, and written answers—using the turnin utility by the same due date & time. The course page (HW link) contains instructions on how to use turnin. For CS 536Q students, please use turnin for all your submissions as with on-campus students. The only difference is that hardcopy submission is not required.

PROBLEM 1 (20 pts)
Read Sections 4.1, 5.1, 5.2, 6.3 from P & D. Read “Congestion avoidance and control” by Van Jacobson. Proc. ACM SIGCOMM ’88, pp. 314–329, 1988. The paper (a version thereof) can found at:
http://www-nrg.ee.lbl.gov/nrg-papers.html
Give a one-page summary and critique.

PROBLEM 2 (30 = 10 + 20 pts)
(a) For congestion control, Method C, perform analysis of its stability following the same steps as carried out for Method D. Show the derivations and explain why one concludes that the protocol is unstable.
(b) For Method D with parameters $Q^* = 100$, $\gamma = 10$, $Q(0) = 0$, $\lambda(0) = 0$, $\varepsilon = 0.2$ and $\beta = 0.5$, calculate the first 25 values of $(Q(t), \lambda(t))$ as time evolves $t = 0, 1, \ldots, 24, 25$, and show the values as points on the 2-dimensional $(Q(t), \lambda(t))$-plane. More precisely, after drawing the points, draw an arrow from $(Q(t), \lambda(t))$ to $(Q(t+1), \lambda(t+1))$, for each pair of consecutive values. Make sure to draw the picture by hand. You may use a drawing program that outputs a ps, gif, or jpg file, but you must still draw the arrows by hand when using the drawing program. Give an interpretation and explanation of the picture that you see. Relate your picture to the time evolution plots in the lecture notes under Method D having the same parameters.

PROBLEM 3 (50 pts)
As a continuation of Problem 3, Assignment IV, extend the UDP client/server application such that the client now takes an additional command-line argument called timeout which it uses to retransmit a request if it hasn’t heard back from the server by the timeout value. The command has the format:

```
% udp-client.bin IP-address timeout command
```

Implement the timer—it periodically goes off after timeout seconds—using the alarm() system call. You need to register a SIGALRM signal handler using sigaction() (do not use signal() to register your signal handler) which is invoked by the kernel each time SIGALRM goes off (i.e., the “signal is raised” in system programming parlance). Upon catching the signal, the command is retransmitted. Let the client print onto the terminal the timestamp, server’s IP address, port number, and the command requested when it first sends out a request, and each subsequent time when a request is retransmitted.

The UDP server needs to be slightly modified as follows. It takes a command-line argument drop-prob which is a real number between 0 and 1. Upon receiving a client request, it throws a coin that comes up heads with probability drop-prob, and if so, discards the client request and resumes waiting for the next request. When discarding a client request, the server prints a message to the terminal which contains the following items: a timestamp of when the client request has arrived (use gettimeofday()), the client’s IP address and port number, and the command requested.

Test the system by executing the client with the command date and timeout value 2, and the server with drop-prob equal to 0.5. Repeat the experiment with drop-prob 0.9. As in Assignment IV, in addition to the code and script files, hand in a screen dump which shows the contents of the two windows where the client and server are executed.
PROBLEM 4 (50 pts)

In yet another twist to Problem 3, Assignment IV, modify the UDP client/server application such that the client takes two additional command-line arguments *packet-spacing* (unit of milliseconds) and *total-count* (an integer) as follows

```
% udp-client.bin IP-address packet-spacing total-count command
```

The new UDP client works by firing off the requested command to the server, sleeps for *packet-spacing* milliseconds (use `usleep()`), then fires off the same command again, sleeps for *packet-spacing* milliseconds, sends the command to the server again, and repeats this process until it has sent a total of *total-count* requests. When sending a request, it takes a timestamp and prints it on the terminal (“sending timestamp”). When it receives the server’s response, it prints the response along with a timestamp showing when it arrived (“receiving timestamp”). On the server side, nothing changes except that the server additionally prints the timestamp when the request arrived, the client’s IP address and port number, and the command requested just before forking a child to handle the task.

Benchmark the system by running a client with *total-count* = 30 and *packet-spacing* = 100 (msec). The command requested by the client should be

```
echo timestamp client-ip client-port request-seq
```

where *timestamp* is the time at which the client sent the request, *client-ip* and *client-port* are the client’s IP address and port number, respectively, and *request-seq* is a sequence number counting from 1, 2, ..., *total-count*. Thus, if any request is dropped by the server—not by artificially discarding requests with some probability, but due to overload—this will be visible on the client side from the gaps in the request sequence number. Repeat the experiment with *packet-spacing* equal to 60, 20, 10, 5, and 1 (msec). Using the script file where the outputs on the client and sender sides are logged, plot a graph (e.g., using `gnuplot`, `xgraph`) where on the x-axis (abscissa) the packet spacing parameter *packet-spacing* is shown, and on the y-axis (ordinate) the average value of the actual observed request spacings on the client side (the difference between the client request’s timestamps). What happens to the actual time interval between client requests as *packet-spacing* is made smaller and smaller? Can you find an explanation?

Plot a second graph where the x-axis is still *packet-spacing*, but the y-axis shows the actual number of responses received (at maximum *total-count*). Are any requests dropped by the server as *packet-spacing* is decreased?

Plot a third graph where the x-axis is *packet-spacing* and the y-axis shows the corresponding average RTT (i.e., “response time”). Is there an increase in the response time as *packet-spacing* is decreased? Explain. What happens if you run four clients—each on a separate machine—who send requests to the same server who resides on a third machine with parameters *packet-spacing* = 10 (msec) and *total-count* = 30? Is there a performance difference across the four clients?

PROBLEM 5 (20 pts)

Suppose the client in Problem 4 above observes that when sending requests at *packet-spacing* = \( \tau \) over some time interval \( T \) (say a time window 1 sec), the loss rate \( 0 \leq c \leq 1 \)—as defined by the ratio of the number of requests dropped to the number of requests sent during \( T \)—starts to increase when \( \tau \) is decreased. If the client’s goal were to send as many requests as possible while keeping the loss rate below some target level \( 0 \leq c^* \leq 1 \) (e.g., \( c^* = 0.1 \) would mean that 10% losses are allowed), design a congestion control protocol aimed at achieving this objective. Argue why your congestion control protocol is expected to perform as desired.