Please type your answers and submit the hardcopy output by 5pm in CS 220 (slide it under the door) 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**

Read Sections 4.2 and 4.3 from P & D.

**PROBLEM 2 [BONUS] (40 pts)**

As a continuation of Problem 3, Assignment V, implement a tunneling application which works as follows. The client takes an additional argument

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

where `router-IP-address` is the IP address of a tunneling router to which the message is to be sent. The message format changes slightly where the first four bytes of the payload contains `IP-address` (the destination or server’s IP address). The tunneling router, call it `udp-tunnel-router.bin`, is a simple stripped down version of `udp-server.bin` which upon receiving a message from the client or server forwards the message to the final destination. The tunneling router performs forwarding by inspecting the first four bytes of the payload and interpreting them as the destination IP address. Before forwarding, it overwrites the first four bytes with the source IP address of the sender so that the receiver can know who the original sender is. The tunneling router should print out a time stamp of when a message is received, the source address and payload. The server is a stripped down version of `udp-server.bin` which just echoes back the `command` received to the original sender through the tunneling router. The server’s message format is as follows: the first four bytes contain the server’s IP address (needed by the tunneling router), a time stamp field of the time when the message was received at the server, and a copy of `command`. The server sends this message to the tunneling router whose IP address it learns from the IP packet header received from the tunneling router. The server is a simple iterative server—not a concurrent server which forks a child as in previous assignments—who parses the request, then formats the aforementioned message, and sends it back to the client through the tunneling router. It still performs probabilistic dropping as in Problem 3, Assignment V, which remains unchanged. The `connect()` based UDP code at all three applications (client, tunneling router, server) should be replaced by the simpler stateless `sendto()` and `recvfrom()` implementation. Repeat the experiments of Problem 3, Assignment V, with client, tunneling router, and server running on separate machines. The logs should clearly show what happens when a message is not dropped by the server, and what happens when a message is dropped.

**PROBLEM 3 [BONUS] (40 pts)**

As a continuation of Problem 5, Assignment V, implement and evaluate the protocol you proposed. The server remains the same and only the client needs to be slightly modified. The client takes two additional parameters

```bash
% udp-client.bin target-loss window-size control-flag IP-address packet-spacing total-count command
```

where `target-loss` is a real number indicating the target loss rate to be achieved by the client and `window-size` is a positive integer that represents the window size (in the space of sequence numbers) over which the current loss rate at the client is calculated/updated. For example, if `window-size = 10` and the client has just sent packet with sequence number `request-seq` (see the `command` specification in Problem 4, Assignment V) and is about to issue a sleep, then the client keeps track of how many responses for the 10 most recent requests (i.e., `request-seq − 1, request-seq − 2, ..., request-seq − 10`) it has received, call this number `x` (note that `0 ≤ x ≤ 10`), calculates `c = 1 − (x/10)` and takes this value as the current loss rate. The new packet spacing—i.e., sleep time—is then determined by your congestion control protocol which uses the current loss rate `c` along with the target loss rate `c^*`. The client should print out the current loss rate, along with the newly computed packet spacing value. `control-flag`
is a binary flag where \( \text{control-flag} = 1 \) means that the adaptive packet spacing control is turned on or active, and \( \text{control-flag} = 0 \) means that it’s turned off (in essence, equivalent to the behavior of the client program in Problem 4 of Assignment V). Thus your code should be modular so that when \( \text{control-flag} = 0 \), it degenerates to a fixed packet spacing behavior.

As a first benchmarking step, run 16 clients (each on a separate host) which send requests to a common server (running on yet another machine) with total-count = 200. Have the control-flag turned off with initial packet spacing set to packet-spacing = 60 (ms). Repeat the run with initial packet spacings set to 20, 10, 5, 1 (ms). For each run, on each host, calculate the average loss rate. Plot the average loss rate, as a function of packet spacing, for each of the client hosts (a single plot containing 16 graphs to facilitate comparability). Discuss what you observe with respect to performance and fairness.

As a second benchmarking step, set the target loss rate \( c^* \) as half the average loss rate in the first benchmarking step when packet-spacing is set to 5 (ms). Repeat the experiment with control-flag turned on and initial packet-spacing set to 60 (ms). total-count remains at 200. For each client, plot a time series that shows on the x-axis the message sequence number (i.e., 200 points) and on the y-axis the current loss rate printed at those sequence numbers. Also, calculate the average loss rate for each client. Do an analogous time series plot where the y axis is the current packet spacing corresponding to the sequence number. Discuss how well your protocol performs with respect to stability (does it jump around), optimality (does it converge to \( c^* \)), and fairness (do some clients get higher loss rates than others). Your protocol may have parameters that need to be fine-tuned. If so, show performance results for a parameter value (e.g., magnitude of step size) that works well and one that doesn’t work well.