CMPT 434
Computer Networks
Assignment Three

Due: Monday March 18th, 6:00pm – late submissions will not be accepted.

Total Marks: 78

Submission Instructions: All assignment submissions must use the Moodle online submission system. Your submission should consist of a single compressed tar file that has been created using “tar -cvzf” on one of the Department’s Linux machines. Your tar file should include, for Part A, your source files, a makefile, a readme file that describes **everything that the marker will need to know** to run and test your code, and a single documentation/design file in either plain text or PDF format. With respect to the documentation/design file – think carefully about what the marker needs to know to evaluate what you have done. In addition to a description of what you have done and justification for your design decisions, you should carefully describe any limitations that your solution has. If the marker discovers limitations that you have not described in your documentation, the marker will conclude that your testing was insufficient and you will be docked marks accordingly. Your tar file should also include a separate plain text or PDF file with your answers for Part B.

PART A (*50 marks*)

In this question the task is to design and implement a protocol for data collection in delay-tolerant, mobile wireless sensor networks. Imagine, for example, a wildlife tracking application in which a number of animals are tagged with wireless sensor nodes. Each sensor node captures information about the movements and environment of the respective animal. Periodically each sensor node generates a data packet containing its recent measurements, and we’d like to deliver this data packet to a base station. The (fixed location) base station will frequently be outside the limited transmission range of a (moving) animal’s sensor node, however. *Epidemic protocols* are one approach to solving this problem. Whenever a sensor node comes within range of another sensor node, the data packets of each may be copied to the other (analogously to the spread of a virus). In this manner, when some sensor node *does* come within range of the base station, it can upload not only its own packet, but potentially those of many other nodes as well.

For this question you will not actually be tagging animals (!), but will instead be simulating such a system using multiple processes. Each of the sensor nodes will be simulated using a separate process. Only the collection of **one** data packet from each sensor node will be considered. We’ll assume that the respective animals move within a square region formed by the (x,y) coordinates (0,0), (0,1000), (1000,0), (1000,1000), with the base station at location (500,500). Each sensor node process *S* should start at a random location within this region, and then, once per second (real time, which might correspond to many minutes of simulated time), change its location by picking a random direction and moving a distance D within the region (if the boundary of the region is hit, "bounce off" in some manner so that the sensor node remains within the region). You will also have a "logger" process that will (i) help with debugging and verifying correctness, (ii) play the role of the base station, and (iii) simplify the task in the simulation of determining what nodes are in range.

Each sensor node process *S* should open a TCP connection to the logger process once per second (initially, as well as after each change of location). *S* should then send the logger process a message giving its ID, the port number it listens on for incoming connections, and its current location. It should be sent back a message indicating whether it is within range of the base station. If it is, the sensor node process should transmit the data packet(s) it has buffered, including for each the ID of the node that originally generated the data packet. The logger process should write this information to `stdout`. If *S* isn't in range of the base station, *S* could then send another message to the logger process asking for the contact information for any other sensor node that happens to be within range of *S*. If there is such a node, the logger process should send to *S* its node ID, and the IP address and port number to use for making a connection to that node. *S* should then open a TCP connection with the node, possibly send/receive data packets on the connection, and finally close the connection. For each exchanged data packet, *S* should also send a message to the logger process giving the IDs of the packet sender, receiver, and original generator. The logger process should write this information to `stdout`. *S* could then repeat this process by sending the logger process another message asking for the contact information for another sensor node within range (if any), until it receives a reply indicating that there are no more such nodes. Finally, the TCP connection to the logger process should be closed.

Your sensor node program should take as command line arguments an integer node ID, a short (at most 10 characters) string that will be the data packet generated by that node, the simulated distance D that is moved every second, the port number it should listen on for incoming connections, and finally the host name and port number to use for making a connection with the logger process. The logger process should take as command line arguments the port number it should listen on for connections, the transmission range T, and the number of sensor node processes.

Your logger process should write a message to `stdout` when the base station has received all packets, giving the elapsed time in seconds from the start of the simulation, as well as a count of the total number of data packet transmissions between sensor node processes (don't include here the data packet transmissions to the logger process), up to that point in time.

Objectives are to achieve a relatively short elapsed time until the base station has received all packets, and yet also a relatively small total number of data packet transmissions between sensor node processes. For your protocol design, think about the factors that could be considered when deciding whether to send a copy of a data packet from a sensor node *S*1 to a sensor node *S2* (e.g., what *S2* already has buffered, who else *S*1 has already sent the data packet to, …). Note that sensor nodes within transmission range can exchange more information than just data packet contents, to make better policies possible. However, for your implementation, start simple; for example, you could start with an implementation that **only** transmits packets to the logger process (no transmissions to other sensor node processes). Once that is working, you could experiment with protocols in which sensor node processes send to each other and are able to buffer some limited number of packets received from other sensor node processes.

10 out of the 50 marks allocated to this question will be for your data collection protocol and its design description including discussion of how you evaluated it (e.g. what parameter choices for D and T did you try, and what numbers of sensor nodes) and discussion of possible improvements that might give a better performance tradeoff between elapsed time and number of data packet transmissions.

Your code should work on the Department's Linux machines. You must use the proper socket API functions for TCP as described for example in Beej's guide to network programming. Note that with our lab configuration, only port numbers between 30000 and 40000 should be used.

Program defensively, checking for possible error conditions, and include informative comments in your code.

PART B

1. (*2 marks*) Give the parameters of a token bucket that limits the long term average rate to 40 Mbps (i.e., 40 x 10^6 bits per second) and the maximum duration of back-to-back packet transmissions to 500 microseconds, assuming a link of capacity of 1 Gbps.

2. (*8 marks*) Suppose that a routing table includes four entries used for forwarding incoming datagrams, with forwarding information as follows:

Entry 1:	address prefix:	10.11.43.128/26
	address of next hop router:	10.11.32.7
	interface:	A
Entry 2:	address prefix:	0.0.0.0/0
	address of next hop router:	10.11.0.73
	interface:	B
Entry 3:	address prefix:	10.11.42.0/23
	address of next hop router:	10.11.47.127
	interface:	A
Entry 4:	address prefix:	10.11.32.0/20
	interface:	A

For each of the following destination IP addresses, state whether an incoming IP datagram would be forwarded on an outgoing interface using a destination link layer address belonging to a next hop router, or a link layer address belonging to the IP datagram's destination host. In the former case, state which next hop router. Make sure you justify your answers.

(a) 10.11.43.127

(b) 10.11.43.160

(c) 10.11.49.123

(d) 10.11.44.222

3. (*2 marks*) A TCP sender's value of *SRTT* is 125 milliseconds, but then a routing change occurs, after which all measured RTTs are 80 milliseconds. How many measurements of the new RTT are required before *SRTT* drops below 100 milliseconds? Assume that a weight of 0.125 is used for the new sample, and a weight of 0.875 for the old value, when updating *SRTT*. Make sure to show how you got your answer.

4. (*2 marks*) Suppose that a TCP connection's RTT is 125 milliseconds except for every *N*'th RTT, which is 500 milliseconds. Note that if *N* is sufficiently large, the 500 millisecond RTT will cause a timeout, since that RTT value will be highly "unexpected" (assume here that *RTOmin* is less than 500 milliseconds), while if *N* is sufficiently small, it won't cause a timeout. What is the **largest** *N* for which the 500 millisecond RTT won't cause a timeout? (Consider only the behavior after there have been many measured RTTs following this pattern, and the initial values chosen for *SRTT* and *RTTVAR* no longer have any significant impact.) Assume that the same weights are used as in question 3 when updating *SRTT*, and weights of 0.25 (new sample) and 0.75 (old value) when updating *RTTVAR*. Make sure to show how you got your answer.

5. (*14 marks*) Suppose that a sender using TCP Reno is observed to have the following congestion window sizes, as measured in segments, during each transmission "round" spent in slow start or additive increase mode:

round	*cwnd* (in segments)	round	*cwnd* (in segments)
1	1	14	35
2	2	15	36
3	4	16	37
4	8	17	38
5	16	18	39
6	32	19	40
7	64	20	41
8	65	21	42
9	66	22	1
10	67	23	2
11	68	24	4
12	69	25	8
13	70	26	16

(a) Identify the transmission rounds when TCP is in slow start mode.

(b) After the 13th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

(c) After the 20th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

(d) What is the initial value of the slow start threshold *ssthresh* at the first transmission round?

(e) What is the value of *ssthresh* at the 18th transmission round?

(f) What is the value of *ssthresh* at the 24th transmission round?

(g) Assuming that a segment loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the new value of *ssthresh* and *cwnd*?