Programming Assignment II

Spring 2013

Due: May 2nd

Academic Honesty Policy

You are permitted and encouraged to help each other through Piazza's web board, in order to discuss and understand concepts learned in class or go deeper in a particular subject. **HOWEVER**, you may **NOT** share source code or hardcopies of source code, or answers to written assignments. Refrain from activities or the sharing materials that could cause your source code to **APPEAR TO BE** similar to another student's enrolled in this or previous years. Cheating will be dealt with severely. Cheaters will be punished. Source code and written answers should be yours and yours only. Do not cheat.

Introduction

In this assignment, you will work on several important netowrk protocols using UDP socket programming.

The assignment can be broadly divided into 2 parts. In the 1st part, you are required to implement a important Reliable Data Transfer (Selective Repeat) Protocol. Besides, you are also required to implement Distant Vector (DV) routing protocol. In the 2nd part, you will combine Reliable Data Transfer and DV together. You will use DV to calculate the shortest path and use Reliable Data Transfer to transmit data through unreliable link. In this way, you can test whether DV gives the right path in real scenario.

Point breakup as follows:

- Part One
  - Selective Repeat (SR): 45 pts
  - Distance Vector (DV): 45 pts
- Part Two
  - Combination: 60 pts

Due dates and Program Submission

This programming assignment is due on **May 2nd**. **No lateness is accepted**.

Before submitting, make sure your program compiles and runs properly on a CS or EE machine, because you will be graded on those machines and if your code doesn't compile, you automatically get a grade of 0.

To submit the assignment, rename the **yourUNI** folder provided with the sample code to your UNI (eg. sb3457). Zip this folder with the same name, and upload this zipped file on CourseWorks.

Ensure that you provide a MakeFile along with your assignment for us to compile.

Do not create subdirectories or files with different names than the ones specified. Incorrect file name and directory structure will be penalized.

The Assignment

Part One - Protocol Implementation

Selective Repeat

Refer to the following link for an illustration of the seletive repeat protocol mechanism:

http://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_6/video_applets/SRindex.html

Implement the sender and the receiver entities in the same class (SRNode) without any loss of generality. **Each process of SRNode can perform both sending and receiving.**

As far as the sender is concerned, it accepts a string as user input and each character of the string is sent as a packet. You are required to print the interaction between the sender and the receiver. The convention for presenting the output format is presented in the **Ouput** section.

Distance Vector

Each node receives a set of it's neighbors and the weight of the link between them as arguments. Once every node in the graph has the knowledge of it's immediate neighbors, start the broadcast at the start node (specified by the user) to compute distance vector routing table of the entire graph.

The goal is to find the shortest distance of each node to all the other nodes in graph according to their routing tables. You are required to print the outcome of the effects of each node's broadcast and finally the distance vector of each node to the other nodes according to the output convention given in the **Output** section.

Implementation Specifics

Name the file **SRNode.java** for the **Selective Repeat** mechanism**.**

Each node implementing the Selective Repeat implementation is instantiated by **SRNode <source-port> <destination-port> <window-size> <time-out> <loss-rate>** where <source-port> refers to the port from the message is sent, the <destination-port> is the port at which messages are received, <window-size> is the length of ACK window, <time-out> is the time-out time in ms and <loss-rate> refers to the rate at which packets are lost - [0,1).

eg. **SRNode 1111 2222 10 300 0.56**

Name the file **DVNode.java** for the **Distance Vector** mechanism.

Each node implementing the Distance Vector mechanism is instantiated by **DVNode <port-number> <neighbor$_1$port> <neighbor$_1$weight> .... <neighbor$_i$port> <neighbor$_i$weight> [last]?**

where <port-number> is the port at which the node listens to, the <neighbor$_i$port> and <neighbor$_i$weight> refers to the port at which the i^{th} neighbor listens to and the weight of the link between the node and the i^{th} neighbor. The **[last]?** term refers to the optional argument which indicates that the currently instantiated node is the last node (i.e. the graph is complete) and this is the node that the starts the broadcasting.

eg. **DVNode 1111 2222 1.2 3333 1.5 4444 0.23**

DVNode 2222 1111 1.2 4444 1.5

DVNode 3333 1111 1.5 4444 0.44

DVNode 4444 1111 0.23 2222 1.5 3333 0.44 last

In the above example node **4444** starts broadcasting which leads to update of the routing tables in all the nodes.

Part Two - Protocol Combination

This part combines the aspects of Selective Repeat with the concept of the Distance Vector mechanism to find the shortest path - the difference being that while each node broadcasts in the distance vector mechanism, now the packets can be dropped. This is where the selective repeat mechanism comes into place to ensure that dropped packets are resent. Also each string constitutes a packet unlike part 1 selective repeat where each character forms a packet.

The goal of this part is same as the distance vector mechanism - to compute the shortest distance between each node and all the other nodes it has access to.

Part two is divided into two subparts:

- Subpart One - Dynamic link weight: The link weight is no longer static. Instead, you are required to change link weight from time to time, and call DV whenever routing table changed.
- Subpart Two - Check correctness of DV: By DV, you have got routing table for each node. Here, you can do a simple test to check whether the routing tables provide the correct path.

Implementation Specifics

Name the file **SDNode.java.**

Each node implementing the combination of the Selective Repeat - Distance Vector mechanism can be instantiated by **SDNode <port-number> <neighbor$_1$port> <neighbor$_1$loss-rate> .... <neighbor$_i$port> <neighbor$_i$loss-rate> [last]?**

where <port-number> is the port at which the node listens to, the <neighbor$_i$port> and <neighbor$_i$loss-rate> refers to the port at which the ith neighbor listens to and rate at which packets are dropped between the node and the ith neighbor. The **[last]?** term refers to the optional argument which indicates that the currently instantiated node is the last node (i.e. the graph is complete) and this is the node that the starts the broadcasting.

Use a common window size of 10, time-out of 300 ms for all nodes.

Note: The weight of the link between node$_i$ and node$_j$ = $1/(1-lossrate(node_i,node_j))$

Hint: For both parts 1,2, you can use the payload of the packets for identification between nodes.

Subpart One - Dynamic link weight:

In this subpart, we are required to change the link weight and call DV whenever table changes.

After the last node (SDNode) starts, the system runs DV. They will be stable after some time (no routing updates). Then you are required to change the weight by command in one node:

change <node$_1$port> <node$_1$loss-rate>.... <node$_i$port> <node$_i$loss-rate>

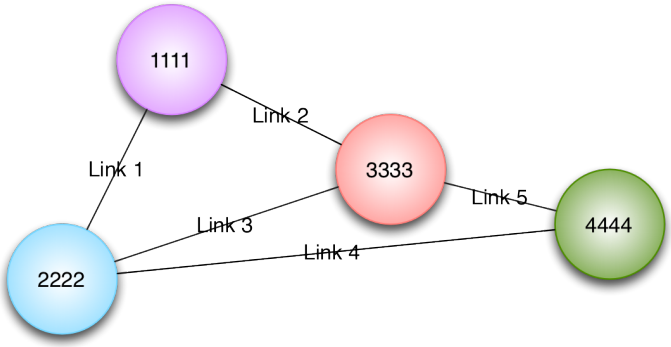
Here you can as many link weights as you like, but if the node in the command is not its neighbour, you just ignore this one (example will be given in the following). Also note that link weight is:

$$w_i = \frac{1}{1 - lossrate_i}$$

Once a node is informed changes on the link, it first send an **link update packet** (contain the new link weight) to all neighbours who shares the link. When its neighbours receive these packets, they update the link weights and send ACK back to the node.

When the node receives all ACK from neighbours that update the link weight, it start DV of the system if its routing table changes.

An example scenario: Given the following picture:



Suppose you input command in node 1111:

change 4444 0.8 3333 0.768

1. Node 1111 check node 4444 and find it is not its neighbour, it ignore this.
2. Node 1111 check node 3333 and find node 3333 is its neighbour, update its routing table. And then it sends the new loss rate of Link2 (0.768) to node 3333.
3. Node 3333 gets new loss rate of Link2 from node 1111. Update the routing table. Send ACK back to node 1111. If node 3333 changes the routing table. It should start broadcasting its new routing table.
4. When node 1111 receives all ACKs to the link update packets it has sent, if its routing table changed. It should start broadcasting its new routing table.
5. After some while, the system will return to stable situation.

Note:

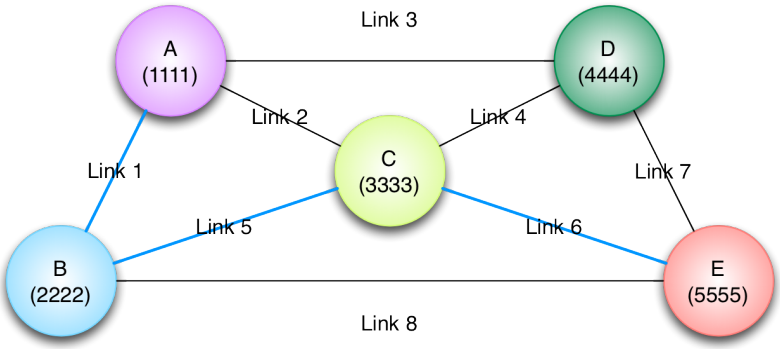
- Different from DV in Part One, here all packets delivery is through unreliable link (including link updating packet, ACK packet, routing table broadcasting packet), using SR to ensure reliable data transfer.
- Pay attention each node broadcast its routing table when its routing table is changed. In some cases, changing link weight does not change the routing table. So there should be no broadcasting in the system.
- If you do SR and DV in part one in the right way, this subpart is not much work. This subpart is actualy a way to test whether you do correctly in part one.

Subpart Two - Check correctness of Distance Vector:

The **weight of each link** is still defined as the expect number of packets sent until one is successfully received (or reciprocal of the success rate).

$$w_i = \frac{1}{successrate_i} = \frac{1}{1 - lossrate_i}$$

Let us see the graph below:



If host A want to send packets to host E through the blue path:

1. Host A is expected to send $N/successrate_1 = w_1 N$ packets to host B when B receives N packets.
2. Host B is expected to send $N/successrate_5 = w_5 N$ packets to host C when C receives N packets.
3. Host C is expected to send $N/successrate_6 = w_6 N$ packets to host E when E receives N packets.

Then totally:

$$(w_1 + w_5 + w_6) \cdot N = \sum_{i \in Path} w_i N$$

If we let all packets sending one by one **in sequence** and assume time needed to send a packet through every link is equal (we run project in one computer by several processes), total delay is then **proportional** to the total weight on the path. This is how we check the rightness of Distance Vector Routing Protocol (DV).

The command for source to send packets to destination is:

send <dst-port> <packet-number>

You type the above command in the source node, <dst-port> is the port for destination node and <packet-number> is the number of packets to be sent.

An example scenario:

1. By DV, each host has its own table which related to the shortest path. Now suppose we want A send to E 100 packets.
2. I type command in node A: send 5555 100
3. Suppose the blue path in Fig 3-1 is the shortest path from A to E.
4. The transmission method uses Selective Repeat (SR). The receiver drops the packets at loss rate of the link (randomization is needed).
5. In order to check the rightness of DV, source host sends packets to all its neighbors, i.e.:

- A prepares to send 100 packets to B, each packet indicates that source is A and the destination is E.
- A prepares to send 100 packets to C, each packet indicates that source is A and the destination is E.
- A prepares to send 100 packets to D, each packet indicates that source is A and the destination is E.

6. A starts sending 100 packets to B, and A also remembers the timestamp $start_B$ when it sends the first packet to B.
7. After B successfully receive the 100 packets from A (In SR, A can know this by the ACK packets), B print the actual number of packets received from A (including packets dropped) and the actual loss rate. A stops sending but waiting. B starts sending according to its routing table (already updated by DV). B knows here it should send to C. Thereby, B sends to 100 packets to C.
8. C does the same as B (after receiving 100 packets, print received number and loss rate. Then send by its routing table).
9. Finally, E recieved the 100 packets from C. E records the timestamp $finish_B$ when it receives the last packet from C. E first does the same print work and then sends the timestamp as a packet back to A (it knows A is the source. **Hint: format your own payload in the packets**). This follows E's routing table.
10. The "$finish_B$" packet will be delivered by several nodes until it reaches A. A print $finish_B - start_B$.
11. After A finished all work with B, it is time for A to do the same to all the other direct neighours (here is C and D). When A finishes all, it should have printed $finish_B - start_B$, $finish_C - start_C$ and $finish_D - start_D$.
12. B is in the routing table of A. Therefore, $finish_B - start_B$ is supposed to be the smallest.

Note:

- Only the source (A in the above example) sends to all direct neighbours, other nodes use their routing table for the next node.
- All packets delivery is through unreliable link (including normal packet, ACK packet, timestamp packet), using SR to transfer data.

Command & Output

Note: All port numbers unless otherwise stated refer to the listening port.

Selective Repeat

Sender - Terminal Commands	
Instantiation	SRNode <src-port> <dst-port> <window-size> <time-out> <loss-rate>
Send Message	send <message>

<message> - a variable-length string

Sender - Print to Terminal	
Send Packet	[<timestamp>] packet-<number> <character> sent
Receive Ack - 1	[<timestamp>] ACK-<number> received
Receive Ack - 2	[<timestamp>] ACK-<number> received; window = [<start>,<end>]
Packet Timeout	[<timestamp>] packet-<number> timeout

<timestamp> - using **Calendar.getInstance().getTimeInMillis()** in Java to get the timestamp.

<number> - a numeric expression {Regular Expression: \b[0-9]+\b} eg. 0, 12, 222

<character> - a character from the message eg. a, 1, f, g

Receive Ack-1 refers to receving the ack but no window advancement occurs, whereas window advancement occurs for Receive Ack-2 where <start> and <end> signify the start and ending packet number of the window.

Receiver - Terminal Commands	
Instantiation	SRNode <src-port> <dst-port> <window-size> <time-out> <loss-rate>

Receiver - Print to Terminal	
Receive packet - 1	[<timestamp>] packet-<number> <character> received
Receive packet - 2	[<timestamp>] packet-<number> <character> received;window = [<start>,<end>]
Send ACK	[<timestamp>] ACK-<number> sent
Discard packet	[<timestamp>] packet-<number> <character> discarded

<number> - a numeric expression {Regular Expression: \b[0-9]+\b} eg. 0, 12, 222

<character> - a character from the message eg. a, 1, f, g

Receive packet-1 refers to receving the packet but no window advancement occurs, whereas window advancement occurs for Receive packet-2 where <start> and <end> signify the start and ending packet number of the window.

The following table shows the parallel terminal view of both the Sender and the Receiver.

Terminal #1 - Sender	**Terminal #2 - Receiver**
SRNode 1111 2222 8 500 0.56	SRNode 2222 1111 8 500 0.65
send abcdefghijk	[1355099776308] packet-0 a received; window = [1,10]
[1355099776284] packet-0 a sent	[1355099776308] ACK-0 sent
[1355099776290] packet-1 b sent	[1355099776309] packet-1 b discarded
[1355099776290] packet-2 c sent	[1355099776309] packet-2 c received;
[1355099776290] packet-3 d sent	[1355099776309] ACK-2 sent
[1355099776291] packet-4 e sent	.....
[1355099776291] packet-5 f sent	
[1355099776293] packet-6 g sent	
[1355099776293] packet-7 h sent	
[1355099776311] ACK-0 received; window = [1,10]	
[1355099776312] ACK-2 received	
[1355099776791] packet-1 timeout	
[1355099776791] packet-1 b sent	
.....	

Distance Vector

DVNode - Terminal Commands	
Instantiation - 1	DVNode <port-number> <neighbor$_1$port> <neighbor$_1$weight> .... <neighbor$_i$port> <neighbor$_i$weight>
Instantiation - 2	DVNode <port-number> <neighbor$_1$port> <neighbor$_1$weight> .... <neighbor$_i$port> <neighbor$_i$weight> last

<port-number> - listening port of the node
<neighbor$_i$port> - listening port of the ith neighbor
<neighbor$_i$weight> - weight of the link between the node and the ith neighbor. **Round to 3 decimal places.**

DVNode - Print to Terminal	
Send Message	[<timestamp>] Message sent from Node <node$_i$> to Node <node$_j$>
Receive Message	[<timestamp>] Message received at Node <node$_i$> from Node <node$_j$>
Print the Updated Routing Table	Refer to the format below

Routing Table Format

[<timestamp>]Node <node$_i$> - Routing Table
Node <node$_j$> -> (<weight>)
.....
Node <node$_k$> -> (<weight>)
Node <node$_x$> [next <node$_a$>] -> (<weight>)

Example
[1355080518127] Message sent from Node 4444 to Node 2222
[1355080518128] Message sent from Node 4444 to Node 3333
[1355080518140] Message received at Node 4444 from Node 2222
[1355080518141] Node 4444 - Routing Table
Node 2222 -> (.8)
Node 3333 -> (.5)
Node 1111 [next 2222] -> (.9)

<timestamp> - refers to the current System time in milli seconds
<node$_i$> - refers to the listening port number of that particular node
<weight> - refers to the weight of the link between the pair of nodes. **Round to 3 decimal places.**
Note: In the output of the routing table **Node <node$_k$> -> (<weight>) -** refers to the neighbor of Node <node$_i$> which has a direction connection. For nodes that are not directly connected **Node <node$_x$> [next <node$_a$>] -> (<weight>)** - refers to the neighbor of Node <node$_i$> which is not directly connected to <node$_i$> but is connected via Node <node$_a$> which is the next node in succession.

Combination

SDNode - Terminal Commands	
Instantiation - 1	SDNode <port-number> <neighbor$_1$port> <neighbor$_1$loss-ratet> .... <neighbor$_i$port> <neighbor$_i$loss-rate>
Instantiation - 2	SDNode <port-number> <neighbor$_1$port> <neighbor$_1$loss-ratet> .... <neighbor$_i$port> <neighbor$_i$loss-rate> last

Combination Subpart One - Dynamic link weight:

SDNode - Terminal Commands	
Change Loss Rate	change <node$_1$port> <node$_1$loss-rate>.... <node$_i$port> <node$_i$loss-rate>

<node$_i$port> and <node$_i$loss-rate> is the port number and new loss rate for the link.

Note: This subpart also requires you to **print all message printed in DV in part one** (print receiving updating packet, sending updating packet and updated routing table). But don't print ACK/receiving/discarding message.

Combination Subpart Two - Check correctness of Distance Vector:

SDNode - Terminal Commands	
Send Checking Packet	send <dst-port> <packet-number>

<dst-port> - port number of the final destination (**Pay attention: not the next node**, it is port of E 5555 in the above example).

<packet-number> - how many packets to be delivered in each term (it is 100 in the above example).

SDNode - Print to Terminal		
Name	Format	Example
Start Sending	[<timestamp>] start <node-port>	[1355099776294] start 5555
Finish Sending	[<timestamp>] finish <node-port>	[1355099776394] finish 5555
Finish Receiving	[<timestamp>] <port> <real-number> <loss rate>	[1355099776394] 1111 189 0.471
Print Time Cost	<src-port> - <next-port> -> <dst-port>: <time cost>	1111 - 3333 -> 5555: 235

SDNode - Print to Terminal (continue)	
Name	Description
Start Sending	When the node start sending the first packet to another node.
Finish Sending	When the node receive the last ACK from another node.
Finish Receiving	When received the last packet in this term from another node.
Print Time Cost	**Only printed by the source,** when it received timestamp from destination

<timestamp> - using **Calendar.getInstance().getTimeInMillis()** in Java to get the timestamp.

<port> - port of the sender node.

<real-number> - how many packets actually sent (including packets dropped).

<loss rate> - actual loss rate (calculated by num_{loss}/num_{total}). **Round the value to 3 decimal places.**

<src-port> - the absolution source (**Pay attention: not the sender node**, actually it is itself because only source print this. It is port of A 1111 in the above example).

<next-port> - port of the next node source choose in this round (each of the neighbour will be selected once in a round, port B, C, D in the above example).

<dst-port> - port number of the final destination (**Pay attention: not the next node**, it is port of E 5555 in the above example).

<time cost> - $timestamp_{finish} - timestamp_{start}$, please check the example above.

Note:

- **Hint**: Each node have to know the following information, you can write them in the payload (**we do not specify payload for any packets this time**):

1. Absolution source: When the final destination receive all packets, it should know to who it should send the timestamp. So this information has to be delivered.
2. Final destination: Each node search for next node by its routing table according to the final destination.
3. Number of packets to send: When receiving, each node only knows sending ACK. But when it ends? This number tells the receiver.

- For above output "Start Sending", "Finish Sending", "Finish Receiving", you don't need to distinguish them from "normal packet" or "timestamp packet". This means you have to output those even for "timestamp packet".
- "timestamp packet" is a single packet, which means nodes finish sending/receiving it when they are sure one of them is sent/received.