COMP416: Computer Networks
Project 3

Due: May 21, 2021, 11:59pm (Late submissions will not be accepted).

Submission of the project deliverables is via Blackboard.

This is an individual project. You are not allowed to share your codes/answers with each other.

Network Layer Analysis and DV Routing Simulator

This project is about the **network layer** of the Internet protocol stack. The objectives are to examine the network layer data, the principles behind network layer services, and routing (path selection). Through this project, you will practice with Wireshark as well as a simplified network routing simulator.

Part-1: Network Layer Analysis

The first part of the project requires you to analyze the local network interfaces and traffic data at the network layer through Wireshark. ICMP traffic will be monitored resulting from the application of a) netstat b) traceroute c) mtr commands. You are required to provide corresponding screenshots of the command and the output (cropped if necessary) for each question.

Network Interface Analysis

You are asked to conduct a preliminary analysis of the network interfaces of your machine. You will use the 'netstat' command to see the status of various network interfaces on your machine. The network statistics (netstat) command is a networking tool used for troubleshooting and configuration, that can also serve as a monitoring tool for connections over the network. Both incoming and outgoing connections, routing tables, port listening, and usage statistics are common uses for this command.
https://docs.oracle.com/cd/E19455-01/806-0916/6ja85399h/index.html
https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/netstat

Please answer the following questions:

1. Use 'netstat -ai' command to view the interface information. Differentiate the interfaces that you see and explain the parameters provided in the output.
2. Use 'netstat -nr' or 'netstat -r' to find the local routing (forwarding) table. Examine the output and explain the following categories from the output: *Destination, Gateway, GenMask, Flags and the meanings of various flags which are set, MSS, Window and 'irtt'.*
3. Open your assigned URL in the web browser. Find out the corresponding TCP socket using the netstat command with the appropriate command line options/arguments.

ICMP Analysis

The Internet Control Message Protocol (ICMP) is a supporting protocol in the Internet protocol suite. ICMP is a companion protocol to IP that helps IP to perform its functions by handling various error and test cases. It is used by network devices, including routers, to send error messages and operational information indicating, for example, that a requested service is not available or that a host or router could not be reached. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute).

You are asked to use traceroute to perform a set of actions as described in the following.

Traceroute is implemented in different ways in Unix/Linux/MacOS and in Windows. In Unix/Linux, the source sends a series of UDP packets to the target destination using an unlikely destination port number; in Windows, the source sends a series of ICMP packets to the target destination. For both operating systems, the program sends the first packet with TTL=1, the second packet with TTL=2, and so on. Recall that a router will decrement a packet's TTL value as the packet passes through the router. When a packet arrives at a router with TTL=1, the router sends an ICMP error packet back to the source. A shareware version of a much nicer Windows Traceroute program is pingplotter (www.pingplotter.com).

The source and destination IP addresses in an IP packet denote the endpoints of an Internet path, not the IP routers on the network path the packet travels from the source to the destination. Traceroute is a utility for discovering this path. It works by eliciting ICMP TTL Exceeded responses from the router 1 hop away from the source towards the destination, then 2 hops away from the source, then 3 hops, and so forth until the destination is reached. The responses will identify the IP address of the router. Since traceroute takes advantage of common router implementations, there is no guarantee that it will work for all routers along the path, and it is usual to see " * " responses when it fails for some portions of the path.

- Start up the Wireshark packet sniffer, and begin Wireshark packet capture.
- Perform traceroute with your assigned URL. (Note that on a Windows machine, the command is "tracert" and not "traceroute".)
- On a Linux machine, you may need to force the traceroute command to send ICMP packets instead of the UDP packets. You may look for this information using '*man traceroute'* and choosing the appropriate flag.
- When the traceroute program terminates, stop packet capture in Wireshark. At the end of the experiment, your Command Prompt Window should show that for each TTL value, the source program sends three probe packets. Traceroute displays the RTTs for each of the probe packets, as well as the IP address (and possibly the name) of the router that returned the ICMP TTL-exceeded message.

4. What is TTL and its significance? Under which layer header can you find the value of TTL?
5. Why is it that an ICMP packet does not have source and destination port numbers?
6. Find the minimum TTL below which the traceroute messages do not reach your particular URL destination.
7. How does your computer (the source) learn the IP address of a router along the path from a TTL exceeded packet?
8. How many times is each router along the path probed by traceroute?
9. Find the route to the IP Address: 18.31.0.200. What is different about the results for this address?
10. What is a Routing Blackhole? Provide a scenario where Routing Blackholes may be used beneficially.

My Traceroute (MTR) is a tool that combines traceroute and ping, which is another common method for testing network connectivity and speed. In addition to the hops along the network path, MTR shows constantly updating information about the latency and packet loss along the route to the destination. This helps in troubleshooting network issues by allowing you to see what's happening along the path in real-time. More details can be found at the links below or other sources from the Internet (you may need to install MTR for Windows and Mac OS).

https://www.exavault.com/docs/help/02-networking/04-mtr

https://www.cloudflare.com/learning/network-layer/what-is-mtr/

11. Use 'mtr *yourURL*' to find continuous statistics of the traceroute. Run the mtr command with three different sets of 4 fields each (you can find the information from 'man' pages) and explain the output.
12. Record the packets using 'mtr yourURL' through Wireshark. What is the difference between the Wireshark capture of traceroute and 'mtr'?

Part-2: DV Routing Simulator

This part of the project involves Distance Vector Routing algorithm and practice with a simplified network simulator. Recall that routing algorithms can be classified according to the type of information kept by each router as either Global where all nodes have the complete topology information (Link State Algorithm) or Decentralized where nodes only know physically connected neighbors and link costs of neighbors. They exchange information with neighbors (Distance Vector Algorithm).

Distance Vector Routing is based on a distributed approach that allows nodes to discover the destinations reachable in the network as well as the least cost path to reach each of these destinations. The least cost path is computed based on the link costs, and each node maintains its own distance vector and each of the neighbors' distance vectors**.**

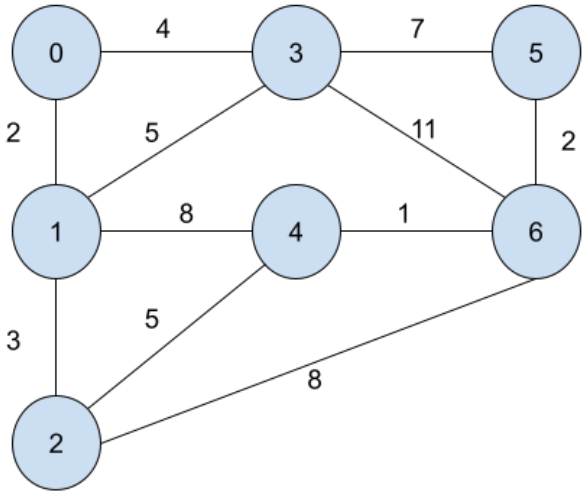

Figure 1 – DVRS default topology

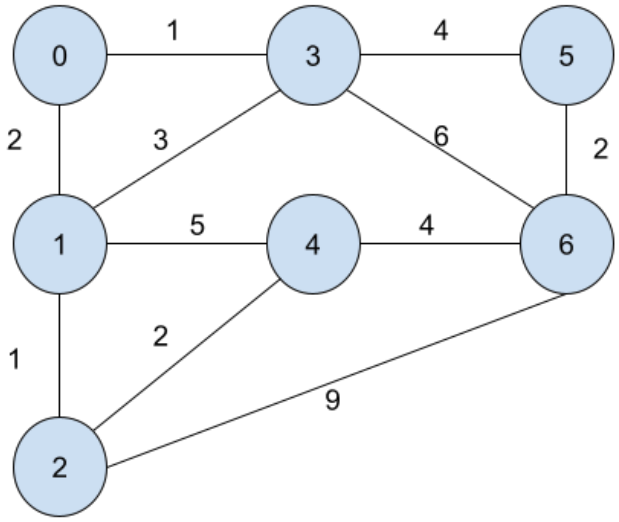

Figure 2 – DVRS topology with different link costs for verification

With this project, you are provided DVRS (Distance Vector Routing Simulator) code framework, and you are asked to design a set of procedures that implements a distance vector routing algorithm for the given network topology as shown in Figure-1. Note that this topology is already set up in the given DVRS code.

After you complete the simulator code and confirm that it works correctly with the given topology, you can update the costs to match the topology given in Figure 2, run the DVRS again and verify that it gives the correct results.

DVRS Code Structure

Project.java [run, read-only]:
This class has the main() method of the project. It only creates a simulator instance and runs it by calling the simulator's runSimulator() method.
Do not modify this file.

DVSimulator.java [use, read-only]:
The distance vector routing simulator is provided here including the simulator variables, the topology, and providing helper methods.

Variables:

- static final int NUMNODES: stores the number of nodes in the system
- Node[] nodes: stores the list of nodes in the network. This is used by the simulator and you don't need to use it.
- static int[][] cost: stores the cost matrix for all edges in the network. If two nodes are not directly connected, the cost is set to 999 (infinity).
- static int[][] neighbors: For each node, it stores the list of neighbors directly connected to that node.

Methods:

- public DVSimulator(): the constructor stores the neighbors and costs as described in the given topology (Figure 1). Only change these values if you want to test the simulator with another topology.

After that, it creates the required number of nodes by creating instances from the class Node.

- **public void runSimulator():** For every packet sent in the network, an event is scheduled. This method runs these events, making sure packets are received to their destinations, and calling the destination node's updateDV method.

- **public static void sendPacket(Packet p)**: use it to send packets from one node to another in the network. It achieves this by scheduling an event for sending the packet.

After all packets are transferred, the system converges to its final state. After that, for each node, we build the final forwarding tables and print them and the distance vectors.

Node.java [modify]:

For each node instance,

- Maintains the neighbors list, costs to neighbors, DV (distance vector) of the node, and the DV of all its neighbors.
- Notifies neighbors of its DV changes.
- Responds to updates from neighbors.
- Can build a forwarding table when required.

Packet.java [use, read-only]:

This class defines a very basic packet with only 3 fields: source, destination, and DV. DV is the distance vector of a node being sent to one of its neighbors. It also provides helper methods to get the source, destination, and DV of a packet, namely; getSource(), getDest(), getDV().

Event.java, EventList.java, EventListImpl.java [skip, read-only]:

This set of classes implements the event system used by the simulator to handle sending and receiving packets.

Part-2.A:

Your task is to complete the missing parts in some methods in the Node.java class, as specified in the code. You don't need to modify any other classes in the project. The simulation environment codes are provided in the Project post.

The goal is to have each node maintain its own distance vector and the distance vector of each of its neighbors. To do this, each node needs to notify its neighbors with every change of its DV (distance vector). Also, when a node receives a packet from a neighbor, it should react accordingly, checking if it needs to update any value in its own DV. If it does update the DV, it needs to notify all of its neighbors again.

Specifically, you need to update the following methods in Node class:

public Node()

The constructor of the node class does not take any arguments, but it should read the following from the DVSimulator class variables:

- The node's specified neighbors.
- The node's specified costs to all other edges. Remember that we use the convention that the cost between any two nodes which are not directly connected (neighbors) equals infinity, represented by the value 999 in code.

After that, the node should initialize its own distance vector, using the variable myDV. As you would guess, initially, myDV values would be identical to the cost values since the node does not have any other information.

Final step in initialization is updating neighbors with the initial DV state, calling the method notifyNeighbors().

public void notifyNeighbors()

For each neighbor to the current node, a packet should be created and sent to that neighbor to update them with the new DV of the current node. The packet would only include the correct source and destination values and the current node's distance vector.

To create a packet, you need to use the provided Packet.java class. To send a packet, you will use the sendPacket() method provided in the DVSimulator.java class. Please, do not create your own sendPacket() method since the provided method makes sure the packet is scheduled to be sent correctly.

public void updateDV(Packet p)

This method is called by the simulator every time a node receives a packet from one of its neighbors. The method takes a Packet instance as an argument. The neighbor's id and DV is already extracted for you from the packet. Also, the neighbor's DV is saved in the neighborDV variable for the id of that specific neighbor.

For every value (destination) i in the newly received DV from the neighbor, you need to check if it gives you a better path to that destination i, using the Bellman-Ford equation. The equation for every new DV received from a neighbor is as follows:

For each destination i

current DV of i := min { current DV of i, cost to neighbor + neighbor's DV to i }

After you check all values in the neighbor's DV, if any value in node's own DV (current DV) has changed, you need to notify neighbors about it. Note that you should notify neighbors at most once for every DV received.

Part-2.B: Optional (Bonus)

In this optional (bonus) part of the project, your task is to implement the algorithm for building the forwarding table. Also, you need to measure how long it takes each node to converge to its final DV values. You should only update the constructor and *updateDV()* methods that you updated in part 2.A.

For building the forwarding table, use the *bestPath* variable, initialize it in the constructor and update it after every DV received from a neighbor (if an update is needed). After the node converges to its final state, simply copy the values in the *bestPath* variable to the *fwdTable* variable. The reason for using the *bestPath* variable is that, in practice, the forwarding table shouldn't be updated that often (with every packet received). It is also much easier to keep track of the best paths to each node with every DV update, instead of doing it in a double loop once after convergence.

public Node()

In the constructor, you need to initialize the *bestPath* variable. While there may be multiple ways to initialize the forwarding table, we require that you initialize it as follows:

- **Best path from *i* to *i* := *i***
- **Best path from *i* to *j* := *j* if *i* and *j* are neighbors**
- **Best path from *i* to *j* := *k* otherwise**
  where k is a randomly selected neighbor of i

Note: you can use the provided *randomNeighbor()* method from the *Node* class to choose a random neighbor of a node.

public void updateDV(Packet p)

1. In part 1, for each value in the DV received from a neighbor, you checked if it gives a better path to a node i, and if there were such a better path, you updated the node's own DV value for node i. In this part, you will add one more step to that case: keeping track of which neighbor gave you that better path by updating the *bestPath* variable.
2. After going through the DV received from a neighbor, it should be checked if <u>any</u> update has happened to the node's own DV. If this is the case, you should increment the counter *numUpdates* <u>once</u>. After convergence, *numUpdates* will reflect the total number of times a node needed to update its own DV, which we use as a measure of convergence.

Demonstration:

In the demo session, you are expected to answer questions on the concepts of Network Layer. You may also be asked about netstat, traceroute and mtr commands with the relevant options you may have used in answering the questions from Part-1.
You are required to demonstrate the working of your Part-2 implementation.

Project Deliverables:

Important Note: You are expected to submit a project report, in PDF format, that documents and explains all the steps you have performed to achieve the assigned tasks of the project. A full grade report is one that clearly details and illustrates the execution of the project. Anyone who follows your report should be able to reproduce your performed tasks without effort. Use screenshots to illustrate the steps and provide clear and precise textual descriptions as well. All reports would be analyzed for plagiarism. Please be aware of the KU Statement on Academic Honesty.

The name of your project .zip file must be <surname>-<KUSIS-id>.zip
You should turn in a single .zip file including:

- Project Report named <surname>-<KUSIS-id>.pdf.

  For Part-1, your report should include the answers to the questions and the corresponding Wireshark screenshots.

  For Part-2, a brief explanation of the implementation of the requirements supported by code snippets.

- Source codes: Containing the source codes of your completed version of part-2.A and part-2.B (optional bonus).

Figures in your report should be scaled to be visible and clear enough. All figures should have captions, should be numbered according to their order of appearance in the report, and should be referenced and described clearly in your text. All pages should be numbered, and have headers the same as your file naming criteria.

If you employ any (online) resources in this project, you must reference them in your report. There is no page limit for your report.

Good Luck!