Programming Assignment 3: Distance Vector

Assigned: November 5, 2015

Due: November 30, 2015

You are to implement distributed asynchronous distance vector routing. Routers send their cost for reaching all nodes within the network to its immediate neighbors. Note that in our discussion of distance vector, we've always referred to the nodes as routers. These nodes are actual sub-networks within an organization's networking infrastructure. For the purposes of this assignment, we will continue to refer to the nodes as routers, and will use the IP addresses of the nodes as the destinations in the forwarding tables, and in the update messages.

The RIP protocol supports both request and update messages. You are only required to implement update messages. Your code should implement periodic updates. The update message will contain a list of IP addresses, and a corresponding cost for reaching each IP address. See Figure 1.

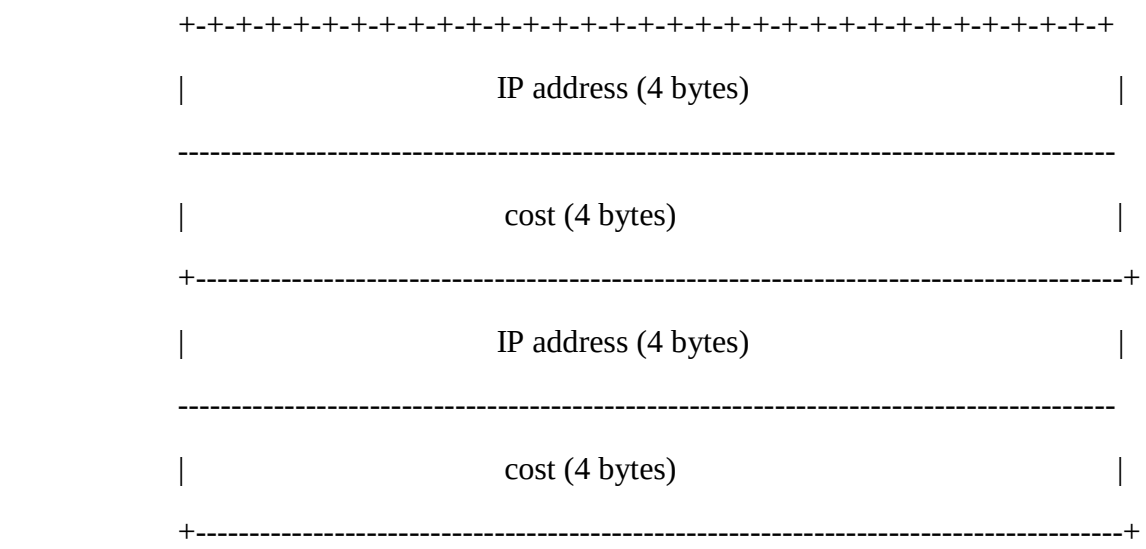

Figure 1: Update Message

The nodes within the network use UDP to send update messages. After receiving update messages from its neighbors, a router will use the Bellman-Ford algorithm to update its routing table. If the router updates any costs in its table, it will send an update message to its immediate neighbors.

Your program should have several components, including: Main, Initialize, Update, Send_Advertisment.

- **Main**- Your main function should accept the following command line parameters: config, portnumber, TTL, infinity, Period.
    - **Config** is a file that contains the network configuration. Each line of the file will be of the following format: <IP address> <neighbor>. The IP address is the address of a node in the network. Neighbor is a Boolean variable which indicates whether the node is a directly connected neighbor or not.
    - **Portnumber** identifies the port that your routing application will use to communicate. Since, multiple student groups may be using the same machines; each student group will be assigned a portnumber. All routers within your network will use the same portnumber.
    - **TTL** is the default time-to-live value for the records in your routing table.
    - Infinity is the default value for indicating that a node is not reachable. You will adapt this value to test how it affects the time that it takes the routers to converge to the actual state of the network. You may test your code with a default TTL of 90 seconds.
    - **Period** indicates the default frequency for sending update messages. You may test your code with a default Period of 30 seconds.
    - **Split Horizon** is a Boolean variable which indicates whether split horizon is to be used when advertising costs to neighbors.

- **Initialize**- Your code should maintain a graph data structure**.** The graph should contain a list of vertices and a list of edges. Given the configuration file**,** the **initialize** function will set the initial values in the graph, as specified by the Bellman-Ford algorithm. Note that you may find a copy of the Bellman-Ford algorithm in the attached document. Initialize should also create the initial routing table.

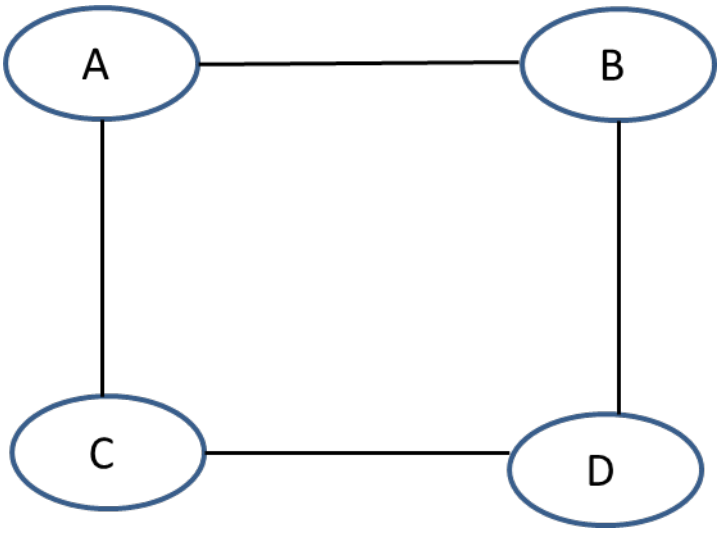

Figure 2: Example Graph

Given, Figure 2 and a source node A, the corresponding configuration file would contain the following lines:
B yes

C yes
D no

Each vertex in the graph should have a label that identifies the vertex and a calculated distance for reaching the node from a specified source (e.g. A in our example). Each edge in the graph has source and destination, and an associated weight. The weight for edges (A, B,), and (A,C) is 1. The weight for edge (A, D) is **Infinity**. Note that A's configuration file does not indicate the relationships between, nodes B and D, B and C or C and D. The Bellman-Ford algorithm would first set the distance for each vertex to Infinity, followed by setting the distance for the source A to 0. Bellman-Ford uses the relax function to determine the distance between the source and the other nodes for which the source has edge costs. For example, Given, Figure 2 below, when initializing the graph at Router 'A', the distance to node B and C would be set to 1, the distance to node D would be set to Infinity.

Before any advertisements from Node A are received, the initial graph at node A is in depicted in Table 1. Since no advertisements have been received from the other nodes, this table assumes that the edges are directed (i.e. no cost information is available for edge (B,A), etc.).

From	**Cost To**			
	A	**B**	**C**	**D**
A	**0**	**1**	**1**	**Infinity**
B	**Infinity**	**Infinity**	**Infinity**	**Infinity**
C	**Infinity**	**Infinity**	**Infinity**	**Infinity**
D	**Infinity**	**Infinity**	**Infinity**	**Infinity**

Table 1: Initial Graph/ Node table at "A"

After Initialize is run, the corresponding routing table for Node A is in Table 2. If the node is a neighbor, your code should set the cost to 1. The cost to reach non-neighbors is **Infinity**, and the nexthop entry for the neighbor will be the neighbor's IP address. The routing table should also contain an entry for the specific router, with cost set to 0 and nexthop, set to the IP address of the router. The TTL value for each record should be set to the default TTL. After setting the initial values of a node's routing table, the node should then output the initial routing table. The initialize function should then call Send_Advertisement to send a router's initial advertisement to its neighbors. The basic structure of a routing table entry is:

- **Typdef struct {**
- **NodeAddr Destination; /* address of the destination */**
- **NodeAddr Nexthop; /* address of the next hop */**
- **Int Cost; /* distance metric */**

- **u_short TTL; /* time to live in seconds */**
- **} Route_entry;**
- **Route_entry RoutingTable[Max_Routes].**

Node	Next Hop	Cost	TTL
A	**A**	**0**	**Default_TTL**
B	**B**	**1**	**Default_TTL**
C	**C**	**1**	**Default_TTL**
D	**Null**	**Infinity**	**Default_TTL**

Table 2: Routing Table for Node A

Figure 2: Example Graph

- **Send_Advertisement() –** Given a routing table, this function will send a message to a router's immediate neighbors, indicating the neighbor's cost to reach other nodes in the network.
- **Update –** Routers will send advertisement messages periodically, as specified by the **Period** command line parameter. The period is set in seconds. The simplest way to manage advertisements is to sleep, for **Period** seconds. Once the time has expired, your **Update** function should do the following:
  - Decrement the TTL in the routing table entries by the specified amount of time.
  - Check for advertisement/update messages from neighbors. Process received messages by updating edge weights in your graph. Run Bellman-Ford to recalculate your routing table. If any routing table entries change, set the TTL for that entry to the default value.
    - Expired TTL – If the TTL for an entry has expired, this means that an advertisement hasn't been received from a neighbor for multiple time windows. When a TTL expires, you should consider the node unreachable, and set the Cost for reaching the node to Infinity,
  - Print the updated routing table.
  - The node then sends an update message to its neighbors.
- **Split Horizon** – In the previous bullets, it is assumed that the routing update/advertisement will contain cost information for all nodes in the network. You are to implement Split horizon. Recall that routers that implement split horizon do not include cost information for nodes that it has learned from neighbors. For example, in Figure 2, if A uses B to get to D, it will not advertise its cost for reaching D to B.
- **Detecting Node Failures** – You will simulate node failures by killing the routing process that is running on a node in your network. After the neighbor hasn't received an update message within TTL time period, the neighbor will assume that the node is no longer reachable and will begin advertising a cost of **Infinity** for reaching the node.
- **Multi-threaded routing application and supporting triggered updates –** The above functions assume a serial application. Given a serial application, the logic for supporting

both periodic and triggered updates may become a bit complicated. Note that triggered updates are sent outside of the period for sending regular periodic update messages. These updates are most likely initiated by changes in a neighbor's routing table. The serial application sleeps for Period seconds, checks for messages, processes updates and then sends out the periodic update.

To support multi-threading, you are to separate the functionality that receives update messages from neighbors from the functionality that processes these messages and updates the graph and routing table. Separating this functionality will allow the main thread to:

- Process messages and pass those messages off to an update thread.
- Call the Update thread every Period seconds.

Note that your threads will have some shared variables (e.g. message, graph, routing table), which will require the use of a mutex and condition variables.

- **Write-up** – Your write-up should describe your graph and routing table data structures. If you've successfully implemented a multi-threaded application, please describe your shared variables, mutex and conditional variables. Your write-up should also discuss and analyze the time it takes for your application to:
  - Establish routes to all nodes during initialization
  - Converge to a steady state after a node goes down. Your analysis should vary the Infinity and describe the effect that this variable has on the time it takes to converge.
    - Also describe how Split Horizon affects convergence.

- Environment –You should run your code on the Burrows machines. Please see the attached list of machines that you are to use while testing your code. You are also to use a port number within the specified range: 53801 – 53825. The AIs will assign a port number to your group. Using an assigned port number will prevent groups from interfering with others while testing their code.