Project 3: Distance Vector

Authors: Kedar Gundlupet Narayana Prasad, Anup Bharadwaj Date: 12/5/15

Introduction:

The distance vector algorithm is iterative, asynchronous, and distributed. It is distributed in that each node receives some information from one or more of its directly attached neighbors, performs a calculation, and then distributes the results of its calculation back to its neighbors. It is iterative in that this process continues on until no more information is exchanged between neighbors. In this project, we implement the distance vector algorithm using C. Each system acts as a router which sends their cost for reaching all nodes within the network to its immediate neighbors. The IP addresses of these systems are the destinations maintained by the program. Please note that the program we implemented accepts only IP addresses, not host name. To test our program, we used several of the servers hosted at IU, with the port number – 65532 as assigned to our team.

Update message format:

In an update message, each node information consumes 8 bytes of data to be sent over the network. The first four bytes is the IP address and the last four is the cost to reach the node. To implement this, we used the bit-shift operators to send this information to the nodes. Inter-conversion of binary to integer and vice versa has been handled by the code.

Implementation:

The network graph:

When the program reads the config file, it internally constructs a graph of the networks with the routers/nodes as the vertex of the graph and stores the information in the form of a matrix which is implemented as a two dimensional array. Each vertex of the graph and its information is now stored in an array of a structure “neighbouring_routers” and they are mapped and maintained accordingly with their array indexes.

```
struct neighbouring_routers{

char ip_addr[15];

int is_neighbour;

};
```

The Routing Table:

After constructing or graph and deciding on the neighbors of a particular node, the program now starts constructing a routing table. It contains the information of the destination, the next hop and a flag indicating if it is the neighbor of a particular node. As and when the nodes advertise new information, the structure is modified accordingly and any changes are further advertised to the neighbors.

Split Horizon:

The program also implements split horizon which can be set or unset by giving a command line parameter. Split horizon is one of the solutions to the count to infinity problem faced by the nodes on detecting a node failure. To enable this in the program, simply pass 1 in the split_horizon command line argument. We the node learns a new cost to reach a particular node from its neighbor, it will maintain another data structure which keeps the information on which among its neighbors sent this information to the current node. Upon sending updates, the node will not advertise this information to that particular node.

Multi-Threading:

Distance vector algorithm supports both periodic updates and triggered updates. When a node sends an advertisement to its neighbors, the neighbors must quickly process the information and advertise any changes further to its neighbors. Along with this, the nodes should also send periodic updates of its view of the network. This cannot be achieved in a serial application. Therefore, it is necessary to implement multi-threading to our application.

With multi-threading, there is an additional responsibility to avoid race conditions of the threads. To do so in our program, we have applied mutex locks on some of our function calls thus ensuring that at any point of time, only one thread has access to the shared variables. The information on the socket is shared by all the threads for sending and receiving the information. Whereas structures like the routing table, graph array and neighbouring_routers structure are protected by mutex locks to prevent any unwanted race conditions and unpredictability of program execution. We did not implement any conditional variables in our application as we felt mutex locks were sufficient to prevent race conditions.

Analysis:

We ran our program simulating various scenarios and the following is our observations:

1) Initialization :
   When we ran the program with a sample config file consisting of four nodes in the network, it took approximately two seconds to establish all routes to all the nodes. However, when the graph of the network is acyclic, we noticed that there is a slight increase in the time taken to establish the routes. In our case of four routers it took approximately a second more than the above scenario.
2) Convergence without split horizon:
   a) With a period of 3 seconds, ttl of 9 seconds, and infinity value of 999, the program took approximately 10 seconds to converge to a steady state after a node failure was detected. It was observed that, the nodes were counting to infinity after a node failure.
   b) By changing the value of infinity to 9999, the program took 20 seconds to converge to a steady state after a node failure. Increase in the value of infinity resulted in longer times for the network to reach a steady state.
   c) When we set the value of infinity to 16, the program quickly converged to a steady state within 9.55 seconds.
3) Convergence with split horizon:

When split horizon was enabled, the count to infinity problem was resolved and the nodes were able to attain a steady state as soon as the node failure was detected. We did not see the nodes counting to infinity in this case.