PROJECT 2
DISTANCE VECTOR PROTOCOL IMPLEMENTATION

BY
AMIT PRATAP SINGH (Person # 50097261)
AMITPRAT@BUFFALO.EDU

A PROJECT REPORT SUBMITTED IN THE PARTIAL FULFILMENT OF REQUIREMENT FOR
CSE 589: MODERN NETWORKING CONCEPTS
COMPUTER SCIENCE AND ENGINEERING
UNIVERSITY AT BUFFALO
FALL 2013

1 Overview

1.1 Distance Vector Algorithm

Distance vector routing algorithm runs on the top of routers in the network and finds out the least cost path to destination. It is one of the most popular distributed routing algorithm. It calculates the complete routing table to all the nodes in network with sharing distance vector with its neighbours only and updating the cost accordingly after receiving updates from other neighbours. It also possesses some problems which were resolved up to some extent using some enhancement in this algorithm such as poison's reverse. However the implementation in this project is limited to general distance vector routing algorithm.

1.2 Problem Statement

This project requires the implementation of simplified version of Distance Vector Routing Protocol. It will run on the top of servers and will talk to its neighbour through UDP connection. Each node will send its forwarding table to its neighbours and also will make changes to its routing table as per the updates received from its neighbouring servers. The implementation should be robust to link changes e.g it should be able to handle if servers does down/comes up in-between. It implements the original Distance Vector algorithm but not the Poison Reverse(Which is the improvement over original algorithm to remove count to infinity problem). In change to original algorithm, it sends the update to its neighbours only on periodic timer expiry or user command(step) to send the force update immediately.

1.3 Functional Overview

1. **Periodic Update**
   - It sends out the periodic update to its neighbours on expiry of periodic timer specified at the beginning of starting program.
2. **Force Update**
   - User can send the force update to its neighbours using 'STEP' command. After STEP command is issued, next periodic update will be sent after entire periodic timeout(as the timer value is reset after STEP command).
3. **Update Link Cost**
   - It allows user to update link cost in-between while running the distance vector algorithm. It changes the cost of the updated link as well as all the other paths which goes through this link. It also allows to update cost to infinity. In some cases, it may help to simulate 'Count to infinity' scenario.
4. **Display Routing Table**
   - It displays the routing table in response to 'DISPLAY' command. The routing table is displayed in sorted(decreasing 'to node' value) order. The simplified version of routing table is immediately shown to user when it updates its routing table while exact format of routing table as mentioned in problem statement can be displayed with this command even though the displayed data is same.

5. **Disable Given Server**
   - It disabled the server id passed to the program in command. It is analogy to the situation when some router goes down. In that case all the routes via that server needs to be routed via some other minimum possible cost path. This command sets the route cost to infinity and stops sending/receiving update to this node. But in addition, this node may come up later so to simulate this if update command is run with Non-Infinity cost then it removed the server from disabled list and start communicating with it again.(Route path may again be updated via this or may be some other node as per Bellman-Ford algorithm minimum cost path).
6. **Simulate Crash**
   - This functionality is simulate the behaviour of server crash. In this case, the server stops sending/receiving data packets to/from its neighbouring nodes and also it sets the cost of all nodes to infinity. The neighbouring host doesn't receive three consecutive updates from this server and they also sets the cost to it as infinity. Apart from it, it also stop taking any input command from user e.g. it stops the STDIN.

7. **Display Packet Counts**
   - It displays the packets count received from last query. After each query, the packet count is reset to zero and it will counting the packet count from this point.

8. **Handle Triple Consecutive Missing Updates**
   - It is to identify if the remote host is dead or alive. In case , it doesn't receive three consecutive updates from its neighbours, it assumes that the neighbour is dead and will set the cost to it as infinity but will keep on sending UDP packets to it. Sometime later, this host may come and then we again update the cost path to it.

1.4 Implementation Details

1. **Periodic Timer Expiry**
   - The periodic timer is handled using 'Select' API timeout feature. In starting it takes the user input and parses the timeout value from it and fills the select timer structure value to this timeout value. Now the Select function waits for either any incoming data on its sockets(including STDIN) or timer expiry. Here I verified the select return type (0 ,-1, or 1) for handling timer expiry or 'fd_set' field value bit set. On timer expiry, it triggers the subroutine to send data packet to all of its neighbours. There is one interesting thing about select timer value that its behaviour is little bit depends upon operating system e.g. in some OS, the value of timeout is updated each time when it comes back to select while others keep the same reduced value(if timeout value was 5 and user input comes at 2 then next timeout value will be 3). Interestingly 'Red hat' doesn't update the timer value e.g. it uses the reduced timer value each time(I found the reference regarding this from beez guide and also some few websites and verified it personally). So I update the timeout value each time when timer expires and also in case of 'step' command.
2. **Update Routing Table**
   - Routing table is updated each time when either it receives any update from its neighbours or user sets it manually. In the beginning Routing Table value set was taken from topology file and hence only neighbours have valid cost value while others are set to Infinity(here its value is 999).After that when it receives any update, it makes the appropriate changes to it. Here it uses the Bellman-Ford algorithm to update the link

cost to minimum cost via any of the neighbour nodes. Each neighbour node does the same thing so the output path to any destination(e.g. 1 -->2 or 1-->3) is minimum. But this algorithm posses the loophole for count to infinity if neighbour nodes keeps on fooling each other for minimum path cost while it has been set to infinity. This implementation has 'Count to Infinity' problem. Also this implementation has cost oscillation when cost of any of the node goes up.

3. **Update link cost**
   - When user manually updates the cost of any path then appropriate changes are made to routing table and then accordingly link cost list is also updated(which is flat one dimensional array to keep the details of cost and path from source to all destinations). When user updates the cost of a path which can be intermediate path for other destinations also in routing table and hence these also should be updated.

4. **Disable Server**
   - For implementing this feature, disable list is maintained .When this command is issued, the server is added to disable list and also the appropriate cost changes are made to routing table. The disable list is maintained to filter the send and receive updates to disabled node(as it is UDP connection so we can't stop remote device from sending data and hence we have to filter it out whatever we want to listen). Also when cost of this path is again set to some valid value using manual update command then this node is removed from disabled list.

5. **Display Routing Table**
   - Routing table is derived from 'link_cost' list which maintains the details of peers with its cost {from_node,to_node,via_node,cost}. The format of routing table display is as follow:
     Destination Peer ID Next Hop ID : Cost
     The routing table is displayed in sorted order of 'to node'. To implements this, it quick sorts the routing table by 'to node' value and then printed to STDOUT. To sort the list, existing 'qsort' API is used.

6. **Packet Count**
   - It displays the total received packet count from last query. This is implemented using two value set { total packet count, last updated packet count} . Here the previous one counts the total number of packets from beginning while the second variable is updated each time when user queries for packet count with current total count and hence each time it displays the difference of both.

7. **Simulate Server Crash**
   - In this scenario, server adds all its neighbours to disabled list i.e. cost to its neighbours is set to infinity and it closes all connections. In addition, it stops taking any input user command. So when its neighbours get the triple timer expiry then they isolate it and hence it is completely removed from network.

8. **Handle Missing Updates**
   - When a server doesn't receive any update from its neighbours for three consecutive timeouts, it sets its cost to infinity but it keeps sending updates to it. For handling the

missing update, on each timer expiry the count of missing updates of each neighbour is increased by one and when it actually received the update from a neighbour the count for particular neighbour is decreased and hence in this way it keeps track of missing counts. And also if it receives any update from its neighbour after two timeout but before third timeout, it will set missing timeout counter for that node to zero. It is to ensure that it counts only consecutive misses but not any three misses.

9. **Handle Packets Reordering**
   - As per the project guidelines the send packet can contain packets in any order and hence the mapping while receiving on server side is need to be done with care. For this implementation, qsort API is used to sort the packets in order of 'to_node' value and handle the same way file updating link cost.

2 File Structure

amitprat_proj2.c

- This files implements the basic functionality for communication two processes over network. It takes the input from user and sets the global variable(periodic timer and topology file) that will be used further in program. It also sets up the server connection that is used to listen to its neighbouring peers for any update. In main server execution function(run_server_part), it listen to both any input from standard input or any update received from server. It can processes the data received on socket and user command in parallel. In 'get argument' routine call, it takes the input argument from user and if the command format is wrong then it prints the suitable message to user terminal. It also deals the basic client connection setup to send the periodic update to neighbours. In the 'handle_cmd_ln_intr ' , it accepts the user command that were input during program execution.

algorithm.c

- This file implements all the functionalities of distance vector algorithm and also the all major functionality of this project. In 'update topology link cost' subroutine, it runs distance vector algorithm over routing table to compute the minimum cost path to its destinations and stores them to 'link_cost' list that can be used further to refer for updated link cost either for displaying user messages or filling out the send packets that will be sent to its neighbours. There are also other subroutine calls almost one for each functionality support.

utility.c

- This file in real doesn't implement any functionality but actually it just support the working of other subroutine that may either be used in main file for parsing/managing user request, server setup or even in algorithm file for supporting checks and display functions.

gprot.h

- It contains the header file. macro, data structure and function declarations.

3 Data Structures

3.1 Macro Values

Macro Name	Value	Description
NUM_SERVERS	5	This is the number of servers for current distance vector implementation. This value can be extended to any higher value.
MY_MAX	999	This is our local infinity value. This value is used to represent infinity for link cost.
TRUE/FALSE	1/0	This is for Boolean value set.
SOCK_FAMILY	AF_INET	This is socket family to be used to making server/client connections.
SOCK_TYPE	SOCK_DGRAM	This is socket type for UDP connections.
SOCK_FLAG	FALSE	This is socket flag in normal case.
MAX_CMD_SIZE	256	This is maximum command string size.
HOST_SIZE	32	This is reference IP address size(it takes IPV4 address size as its reference).
MAX_CMD_LN_VAR	4	It is max limit for command line variables.
MAX_BUF_SIZE	1024	This the send/receive buffer size.

3.2 Structures/Unions:

3.2.1 Structure Name : LINK_COST_STRUCT

Purpose : It stores the link cost related details for each peer. Initially it has only valid cost and via node value set for its neighbouring nodes and rest have infinity as default cost and source node as default via node. This structure is continuously gets updated by using routing table updates using

Bellman-Ford algorithm to track the minimum cost path. The printed routing table in STDOUT uses these value set.

Date Filed Name	Date Filed Type	Description	Default Value
From_node	Unsigned Short	This is the source node id.	Source_node
To_node	Unsigned Short	This is destination node ID.	Destination_node
Via_node	Unsigned Short	This is intermediate node through which we can reach from source to destination in minimum possible cost. Initially it is set to source node and later it gets updates as per updates from neighbours.	Source_node
Cost	Unsigned Short	This is the current cost value for above source/destination set via mentioned via node.	INFINITY(999)

3.2.2 Structure Name : PEER_STRUCT

Purpose : It stores the full peer details to track the peer id/port and IP address details. These details are used while identifying source server id when we receive packet from any of the peers. Also these details are used in displaying user messages when we send/receive packet to its neighbours. Initially topology file is read and peer details are filled in this structure.

Date Filed Name	Date Filed Type	Description	Default Value
Peer_id	Unsigned Short	Server ID.	0
port	Unsigned Short	Server Port Number	0
Cost	Unsigned Short	Cost from source node to destination node. Initially it is read from topology file and rest are set as INFINITY	0(in case of peer_id == server_id) other wise INFINITY.
reserve	Unsigned Short	Not Used	0
Peer_addr	char	It stores the server IP address.	NULL

3.2.3 Structure Name : TOPOLOGY_STRUCT

Purpose : This is the initial structure which stores the details of all servers read from topology file. The topology file is read only once and all its details are stored in this structure for future references. It stores the server details as well as all its peer details.

Date Filed Name	Date Filed Type	Description	Default Value
Num_servers	Unsigned Short	Total number of servers.	0
Num_peers	Unsigned Short	Total number of neighbour peers	0
Server_id	Unsigned Short	ID is the server	0
Server_port	Unsigned Short	Server Port	0
Server_addr[HOST_SIZE]	Character array	Server address	NULL
Peer_detail[MAX_NUM]	Peer_struct	It stores the details of all servers including neighbours	NULL

3.2.4 Structure Name : SENDER_PACKET_STRUCT

Purpose : This is the send packet structure. All details are filled in this structure while sending data over UDP socket. When complete structure is filled, the structure is packet into bytes and sent to neighbours. Neighbours unpacks the byte data received and then cast it back to this structure type and can access directly data from structure.

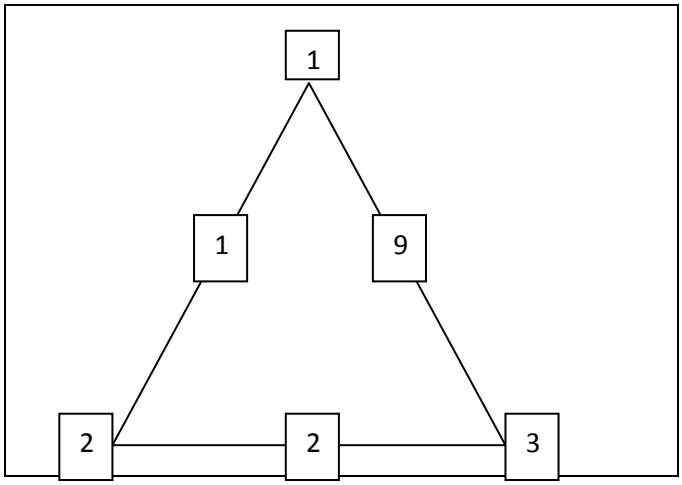
Date Filed Name	Date Filed Type	Description	Default Value
Num_update_fields	Unsigned Short	Total number of update fields included into the packet. It is the count of neighbour details that recipient should read upto.	0
Server_port	Unsigned Short	Port of the sender node.	0
Server_ip[HOST_SIZE]	Character array	Server IP address	NULL
Peer_detail[MAX_NUM]	Peer struct	It holds the data about all neighbours that is needed to sent over UDP socket to its neighbours.	NULL

3.2 Arrays

Routing_table[MAX_NUM][MAX_NUM]	Integer	It maintains the cost from source node to destination node through every via node in row-major format (e.g each row represent the 'to node' cost while each column represents the via node.
Identify_missing_update_node[MAX_NUM]	Boolean	It is a boolean array to keep track of missing update node so that it doesn't always perform the all steps for 'set cost to infinity' operation again and again.

4 Observation

4.1 Triangle Topology



Simulation: Each server is supplied with a topology file and timeout period when starting the program. As soon as program starts, it reads the topology file and start sending the cost details of its neighbours to other neighbouring nodes. Each node start computing the minimum cost path as per Bellman-Ford algorithm and this algorithm converges at some minimum cost path.

Result:

- **Normal Run (with display command)**

Server 1:

```
proj2$_>
 Updates Received from Server :2
 1 -(1)-> 1: 0
 1 -(1)-> 2: 1
 1 -(2)-> 3: 3
 1 -(1)-> 4: 999
 1 -(1)-> 5: 999
proj2$_>
 Updates Received from Server :3
 1 -(1)-> 1: 0
 1 -(1)-> 2: 1
 1 -(2)-> 3: 3
 1 -(1)-> 4: 999
 1 -(1)-> 5: 999
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
 Updates Sent to Server : 128.205.36.24:5556
proj2$ >
```

Server2:

```
proj2$_>
 Updates Received from Server :1
 3 -(2)-> 1: 3
 3 -(3)-> 2: 2
 3 -(3)-> 3: 0
 3 -(3)-> 4: 999
 3 -(3)-> 5: 999
proj2$_>display

 Current Routing Table Status:
 1   2 : 3
 2   3 : 2
 3   3 : NIL
 4   3 : INF
 5   3 : INF
Display SUCESS
proj2$ >
```

Server3:

```
proj2$_>lay

 Current Routing Table Status:
 1   2 : 1
 2   2 : NIL
 3   2 : 2
 4   2 : INF
 5   2 : INF
Display SUCESS
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>
```

This is the saturation status after which node1 and node 2 has chosen least cost path to each other via node 2. Initially when server was started the cost was via source node (i.e. 1-->3 : 9).

Initial cost status for Server3:

```
proj2$_>
 Updates Received from Server :1
 3 -(3)-> 1: 9
 3 -(3)-> 2: 2
 3 -(3)-> 3: 0
 3 -(3)-> 4: 999
 3 -(3)-> 5: 999
```

- **Packet Count**

```
proj2$_>pack
 Updates Received from Server :1
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>ets

 Number of distance vector packets received :166
Packets SUCESS
proj2$_>packets

 Number of distance vector packets received :0
Packets SUCESS
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>packe
 Updates Received from Server :3
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>ts

 Number of distance vector packets received :1
Packets SUCESS
```

- **Step Command**

```
proj2$_>step
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>

 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
Step SUCESS
proj2$_>step

 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
Step SUCESS
proj2$_>
 Updates Received from Server :3
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>
```

- **Disable Command**

Server2

```
proj2$_>
 Updates Received from Server :3
 2 -(3)-> 1: 11
 2 -(2)-> 2: 0
 2 -(2)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>
```

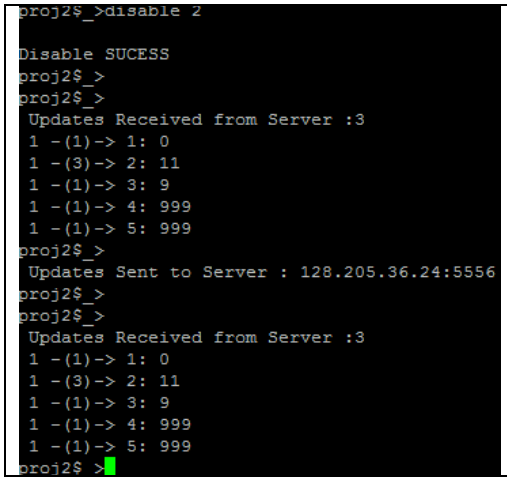
Server3

```
proj2$_>
 Updates Received from Server :1
 3 -(3)-> 1: 9
 3 -(3)-> 2: 2
 3 -(3)-> 3: 0
 3 -(3)-> 4: 999
 3 -(3)-> 5: 999
proj2$_>
```

Server 1

```
proj2$_>
 Updates Received from Server :3
 1 -(1)-> 1: 0
 1 -(3)-> 2: 11
 1 -(1)-> 3: 9
 1 -(1)-> 4: 999
 1 -(1)-> 5: 999
proj2$_>
 Updates Sent to Server : 128.205.36.24:5556
```

Disable Command was passed at Server 1 to disable server 2. Hosts comes back to normal cost scenario if the update command is run as server 1 to set 1 to 2 cost to previous one.

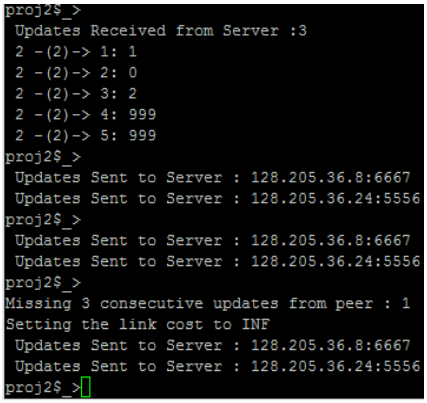


After disable command is passed, server1 stops sending and receiving packets to server2 and hence Server2 after 3 missing update sets its cost to infinity. Now if we pass 'update 1 2 1' & 'update 2 1 1' on server 1 and server 2 respectively, both reset their costs to previous level (as shown above in normal condition).

- **Count to Infinity**

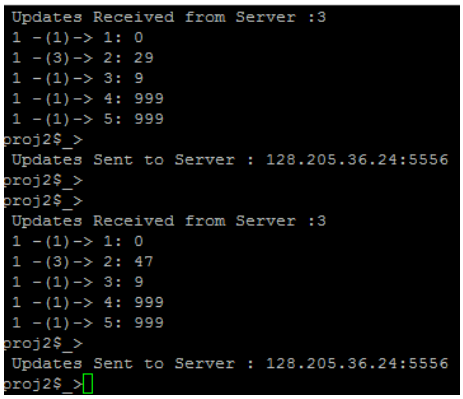
Disable server 2 from both server1 and server 3. Now both the servers will starts computing cost to server 2 via each other and it result to count to infinity situation. This cost oscillation ends when both servers reaches at their infinity cost(here 999).

Server 2

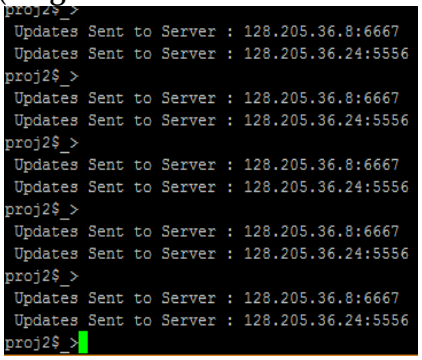


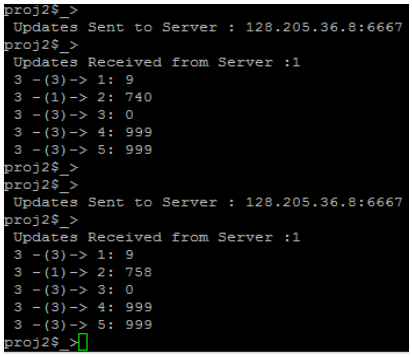
Server 3

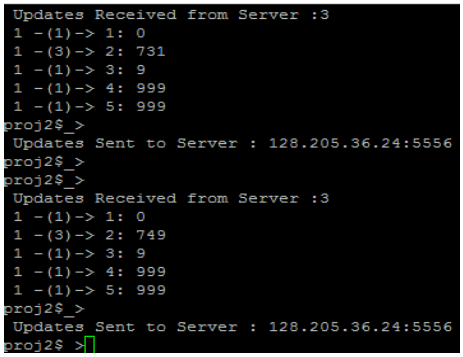
Server1



(Stage at some later moment for all servers)



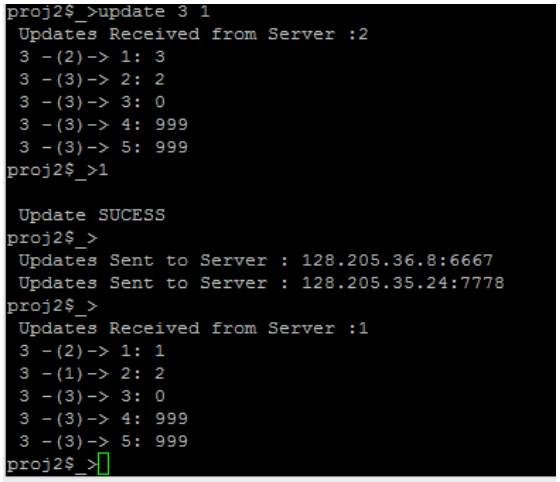




- **Update Command**

Update the cost of 1-->3 link to 1.
Command passed:
Update 1 3 1 (at server 1)
Update 3 1 1(at server 3)

Server 3



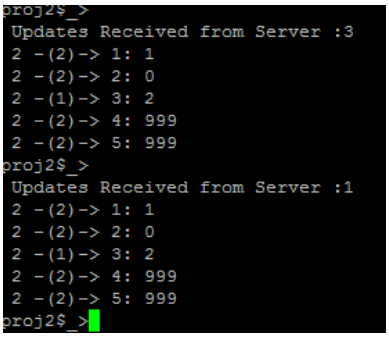
Server 1

Modified Cost Status:

Server 3

Serer 1

Serer 2



- **Simulate Crash**

(Servers are in same situation after above issued update command for link 1 to 3 as 1.)
Crash Command is issued at Server2

```
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 1
 2 -(2)-> 2: 0
 2 -(1)-> 3: 2
 2 -(2)-> 4: 999
 2 -(2)-> 5: 999
proj2$_>crash

Crash Simulate SUCESS
proj2$_>
proj2$_>dis
proj2$_>play

proj2$_>sdf
sdf
```

Server 1 and 2 handling missing updates from server 2

```
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Received from Server :1
 3 -(3)-> 1: 1
 3 -(1)-> 2: 2
 3 -(3)-> 3: 0
 3 -(3)-> 4: 999
 3 -(3)-> 5: 999
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.35.24:7778
proj2$ >
```

```
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.35.24:7778
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>
 Updates Received from Server :3
 1 -(1)-> 1: 0
 1 -(3)-> 2: 3
 1 -(1)-> 3: 1
 1 -(1)-> 4: 999
 1 -(1)-> 5: 999
proj2$_>
```

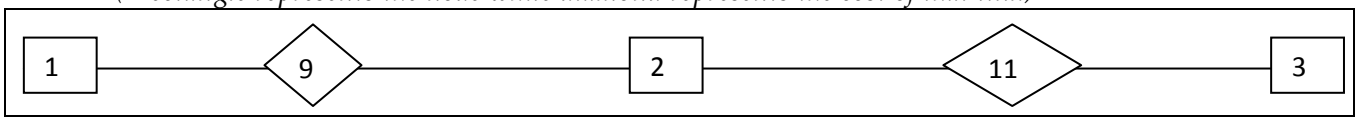
Server 1 and 3 oscillates in count to infinity situation

```
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Received from Server :1
 3 -(3)-> 1: 1
 3 -(1)-> 2: 16
 3 -(3)-> 3: 0
 3 -(3)-> 4: 999
 3 -(3)-> 5: 999
proj2$_>
```

```
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>
 Updates Received from Server :3
 1 -(1)-> 1: 0
 1 -(3)-> 2: 15
 1 -(1)-> 3: 1
 1 -(1)-> 4: 999
 1 -(1)-> 5: 999
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>
```

4.2 Line Topology

(Rectangle represents the node while diamond represents the cost of that link)



- **Normal Run**

Server 1

```
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Received from Server :2
 1 -(1)-> 1: 0
 1 -(1)-> 2: 9
 1 -(2)-> 3: 20
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
```

Server2

```
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
```

Server3

```
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Received from Server :2
 3 -(2)-> 1: 20
 3 -(3)-> 2: 11
 3 -(3)-> 3: 0
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Received from Server :2
 3 -(2)-> 1: 20
 3 -(3)-> 2: 11
 3 -(3)-> 3: 0
proj2$_>
```

- **Update Command**

Server 2

```
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>update 2 3 INF

 Update SUCESS
proj2$_>
 Updates Received from Server :3
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 999
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 999
```

Server3

```
proj2$_>UPDATE 3 2 I
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>NF

 Update SUCESS
proj2$_>
 Updates Received from Server :2
 3 -(2)-> 1: 999
 3 -(3)-> 2: 999
 3 -(3)-> 3: 0
proj2$_>
```

- **Step Command**

```
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>step

 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
Step SUCESS
proj2$_>step

 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
Step SUCESS
proj2$_>
 Updates Received from Server :3
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>step

 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
Step SUCESS
proj2$_>
```

- **Disable Node**

Disable Command is issued at server 2 and 3.

Serve2

```
proj2$_>display

 Current Routing Table Status:
 1  2 : 9
 2  2 : NIL
 3  2 : INF
Display SUCESS
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 999
proj2$_>
```

Server 3

```
proj2$_>isable 2

Disable SUCESS
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>display

 Current Routing Table Status:
 1  2 : INF
 2  3 : INF
 3  3 : NIL
Display SUCESS
proj2$_>
proj2$_>
```

- **Handle Missing updates(in this case server 2 dies)**

```
 Updates Received from Server :3
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
 Updates Sent to Server : 128.205.36.8:6667
 Updates Sent to Server : 128.205.36.24:5556
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
nickelback {~/Fall_2013/amitprat/MNC_PROJ2/new_topology
_files/Server2} >
```

```
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>display

 Current Routing Table Status:
 1  2 : 20
 2  3 : 11
 3  3 : NIL
Display SUCESS
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>display

 Current Routing Table Status:
 1  2 : INF
 2  3 : INF
 3  3 : NIL
Display SUCESS
```

```
 1 -(2)-> 3: 20
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>display

 Current Routing Table Status:
 1  1 : NIL
 2  1 : INF
 3  2 : INF
Display SUCESS
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
```

- **Crash**

```
proj2$_>
 Updates Received from Server :3
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>
 Updates Received from Server :1
 2 -(2)-> 1: 9
 2 -(2)-> 2: 0
 2 -(2)-> 3: 11
proj2$_>crash

Crash Simulate SUCESS
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>
proj2$_>display
```

```
 3 -(3)-> 3: 0
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>display

 Current Routing Table Status:
 1  2 : INF
 2  3 : INF
 3  3 : NIL
Display SUCESS
proj2$_>
```

```
 1 -(1)-> 2: 9
 1 -(2)-> 3: 20
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
Missing 3 consecutive updates from peer : 2
Setting the link cost to INF
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>
 Updates Sent to Server : 128.205.35.24:7778
proj2$_>display

 Current Routing Table Status:
 1  1 : NIL
 2  1 : INF
 3  2 : INF
Display SUCESS
proj2$_>
```

References:

- Beez Guide for Network Programming
- Computer Networking - A top down approach