University at Buffalo
The State University of New York

Distance Vector Routing Protocol

By : Santosh Kumar Dubey

Date : MAY - 02 - 2014

Table of Contents:

1. Introduction **3**

2. Implementation **4**

2.1 Data ‘struct’ Declaration 6

i) Define for COST INFINITE: 6

ii) Data structure of the server: 6

iii) Data structure of cost table : 7

iv) Data structure of routing table : 8

v) Data structure of the update message : 9

3. Algorithm detail **10**

4. Function Call Graph & working **12**

1. load_config() 13
2. start_server() 13
3. read_messages() 13
4. send_update_message() 13
5. read_commands() 14
6. cmd_step() 14
7. cmd_packets() 14
8. cmd_crash() 14
9. cmd_disable() 14
10. cmd_update() 14
11. update_distance_vector() 15
12. Bellman_Ford_Equation() 15

5. Observation & Analysis of routing loop **16**

6. Project Snapshots **23**

7. References **26**

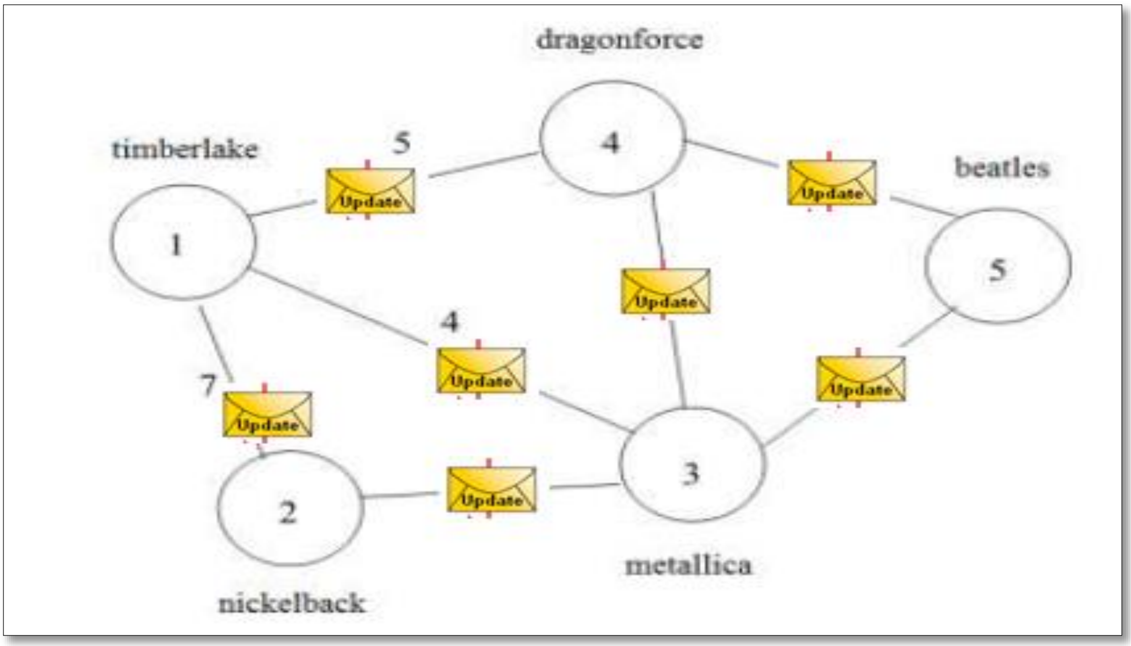
Santosh Kumar Dubey
Person # 5009 7059

1. Introduction

- Each router/server knows the cost of its directly connected server/router (Neighbors).
- A node sends the routing updates to its neighbors periodically.
- If node updates their cost then all connected node in the network updates their routing table eventually.
- If new node is connected then it advertise to their neighbors.

Some important features of the Distance vector routing:

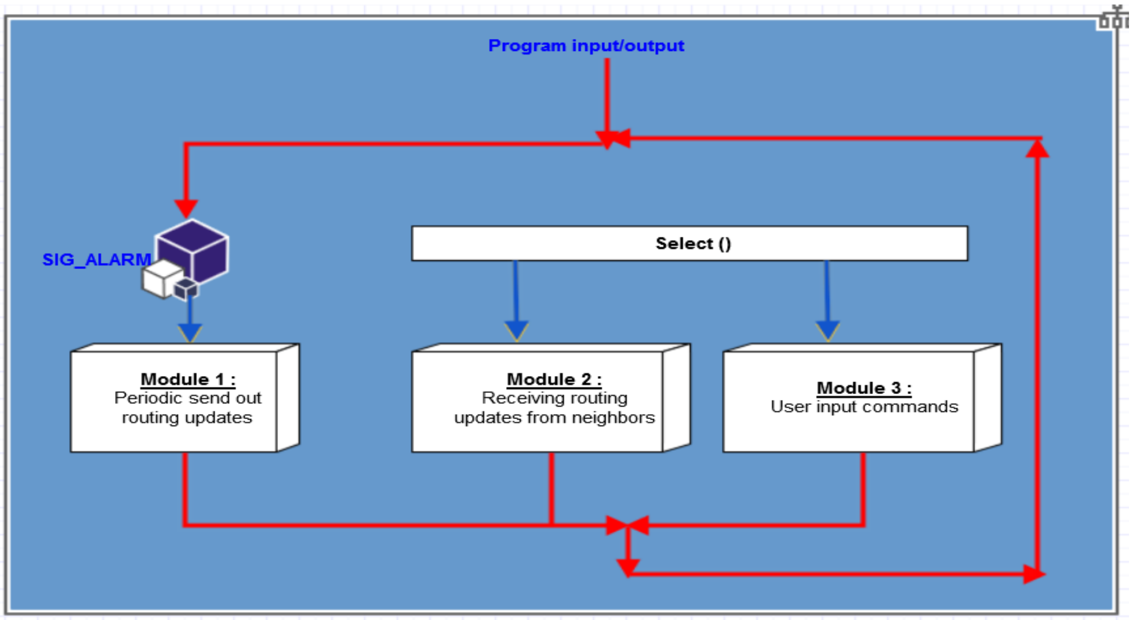
1) **Periodic Updates:** Updates to the routing tables are sent periodically.
2) **Triggered Updates:** If a cost of the link changed, a server/node sends out an update immediately.
3) **Full Routing Table Update**: Some distance vector routing protocol send their neighbors the entire routing table if some changes happened.
4) **Route invalidation timers:** Routing table entries are invalid if they are not refreshed. In this project a typical value is to invalidate an entry if no update is received after 3 update periods.



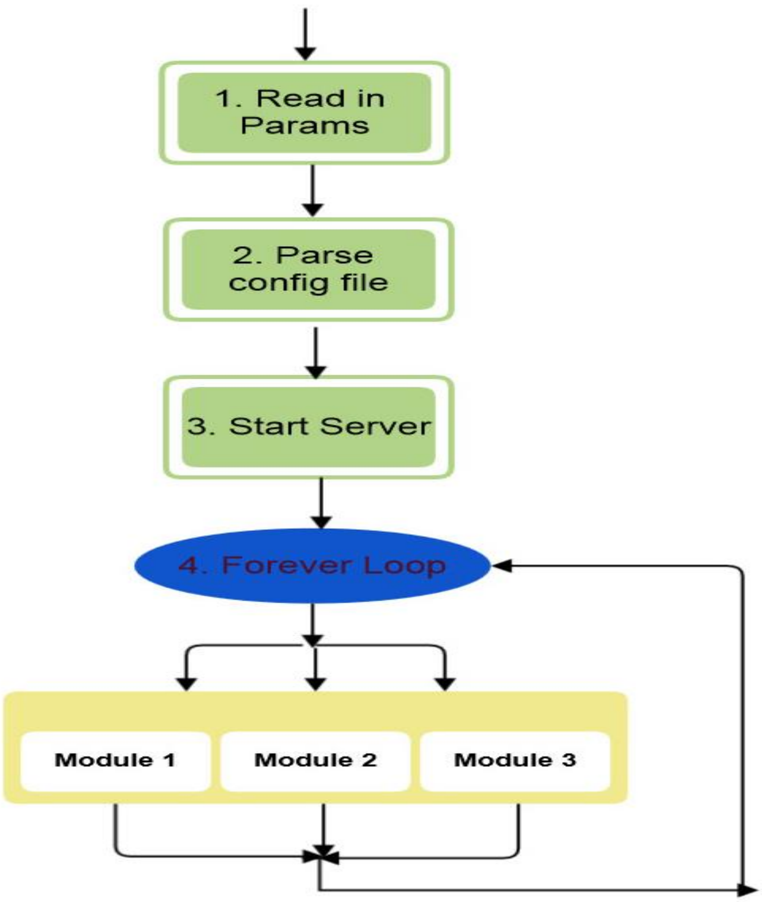
Network Topology

2. Implementation

Overview of this project



Main flow chart:



974 **alarm_handler(int sig) {} :**

To implement the timer in order to send out message in every time interval (second) one signal alarm handler (alarm_handler) have been implemented. This handler will keep on check the servers link in every 3 time unit. If any of the server disappeared from the network or crashed then this alarm handler will send message 3 times to verify whether the server is in network. If the corresponding server do not get any update from its neighbor then it will automatically assign that disappeared as infinity "inf"

In that case we only disable the path and assign the cost infinity, we do not close the path.

If later on that disappeared server joins the network again then we keep the links open and exchange the message based on the time interval period defined by the user but to make the things simple, the cost will be defined as infinity only.

712 **read_commands(int fd){} :**

After we finish reading the config file we store them in a buffer and execute all commands based on config file. We always remove last new line of the input buffer. For some reason if server crashed then we do not response any command inserted by the user.

Some important validation :

i) If server crashed, do not response any commands.
ii) Case sensitivity of the command (ex : Display, display)
iii) Wrong inserted commands.
iv) Checking buffer and lines
v) Displaying success result.

2.1 Data 'struct' Declaration

i) Define for COST INFINITE:

The cost of infinity is defined is "0x7FFF" because 'infinite' should be a large integer, and must not exceed 2-bytes long, so chosen "0x7FFF" as the cost infinite.

```
/* infinite cost define, the largetst integer of "short int" */
#define COST_INFINITE      0X7FFF
```

ii) Data structure of the server:

- Firstly, a "struct" have been defined to store all informations of one server.

```
/* profile for one server */
struct server {
        int id;                 /* server id */
        char ip[16];            /* server ip */
        char hostname[NI_MAXHOST];    /*hostname*/
        unsigned int uip;
        int port;               /* server port */

        int isneighbor; /* 1: is neighbor      0: not neighbor */
        int cost;               /* path cost */
        int forwardid;
        int pathclose;    /* 1: path is closed       0: path not close */

        time_t last_time;      /* last update time */
};
```

All "struct server" includes information in three parts :

- Part 1: Basic info such as IP, port, hostname and so on.
- Part 2: Path info, such as is neighbor, cost and so on.
- Part 3: The last time, we communicated.

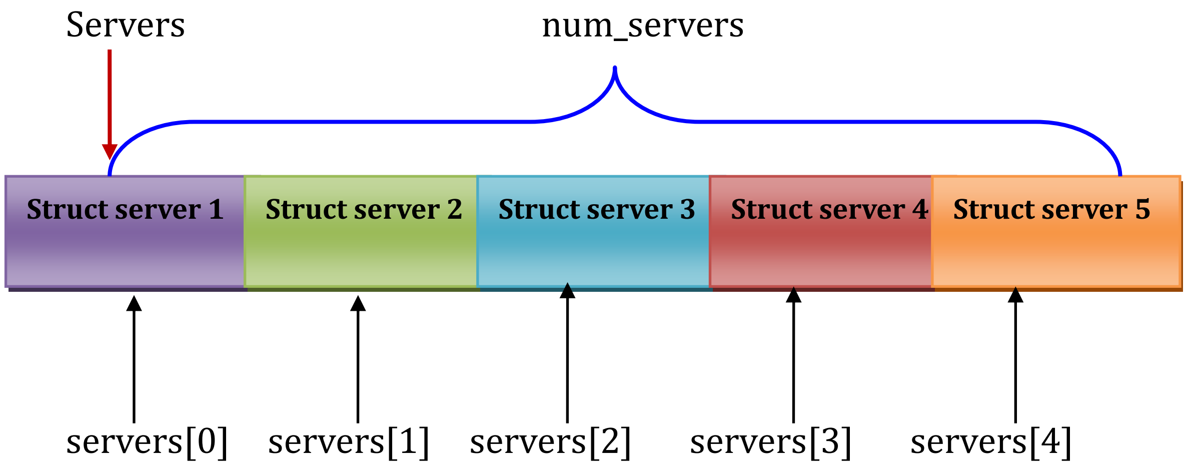
- Secondly, The data-structure "Array" have been used, which dynamically allocate memory to store all the servers with their respective information;

```
int num_servers = 0;       /* number servers */
/* info for all servers */
struct server *servers = NULL;
```

The Number of servers are stored in a global param "num_servers" .

- Thirdly, sort the array by server-id, from small to big.
  For example, in our test case, we have 5 servers, the memory view and how to access the servers will be something like bellow.

Whatever the input sequence is, sort them by server-id, from small to big.



iii) <u>Data structure of cost table :</u>

Actually here we defined cost and forward server-id together with other attributes of the server.
The start server of the cost is always current server.

```
/* profile for one server */
struct server {
        int id;              /* server id */
        char ip[16];         /* server ip */
        char hostname[NI_MAXHOST];    /*hostname*/
        unsigned int uip;
        int port;            /* server port */

        int isneighbor; /* 1: is neighbor      0: not neighbor */
        int cost;            /* path cost */
        int forwardid;
        int pathclose;    /* 1: path is closed       0: path not close */

        time_t last_time;     /* last update time */
};
```

- All servers are stored in an array servers, the number of servers is stored in a global param “num_servers”.
- When we load config file, we allocate memory and initialize the server, then read the basic information from config file.
- We store the attributes like last time when we had received message, this can be used for recording and to verify whether the server is lost for three time interval or not.

iv) Data structure of routing table :

- This is defined as 2D-array to store the distance vector.
- Distance_table[i][j] means the distance from the i(th) server to the j(th) server. (Note: x and y here are indexes of array, not server id).

```
int **distance_table = NULL;        /*Distance Table*/
```

distance_table   distance_table[2][2]   (distance from server 2 to server 2)

D[1][1]	D[1][2]	D[1][3]	D[1][4]	D[1][5]
D[2][1]	D[2][2]	D[2][3]	D[2][4]	D[2][5]
D[3][1]	D[3][2]	D[3][3]	D[3][4]	D[3][5]
D[4][1]	D[4][2]	D[4][3]	D[4][4]	D[4][5]
D[5][1]	D[5][2]	D[5][3]	D[5][4]	D[5][5]

- Now after this the main question comes in mind, How should we get the distance from server-id x to server-id y?
- So here just call function get_server_index() to transfer server-id to the index of an array servers.

For example

i = get_server_index(x); j = get_server_index(y),

Then we know that distance_table[i][j], will be the cost from server-id x to server-id y.

```
/* find server index by server id */
int get_server_index(int serverid)
```

❖ **Why we use an array to store all the servers?**

a) Since one server is communicating with other servers, so we need to store all information of other servers.

b) We used an array to store all servers, because it is very simple and efficient to find a particular server from an allocated memory;
For example, if we know the server-id is 3, then it is very easy to find out because the index memory allocation inside an array will be server[2].

c) We allocate and free memory for all the servers dynamically, because we do not know how many server it will be, before we read from the config-file.

v) Data structure of the update message :

I have defined it in two parts (for the update message), one is header of the message, and the other is the field of the message.

```
/* define for message */
typedef struct {
        short num_fields;
        short port;
        unsigned int ip;
} __attribute__((packed)) head_t;
typedef struct {
        unsigned int ip;
        short port;
        short reserved;
        short id;
        unsigned short cost;
} __attribute__((packed)) field_t;
```

- **How this datastructure is working?**

a) The above two structures "head_t" and "field_t" is for sending and receiving messages.

b) When we need to send and receive messages, so firstly we need allocate a buffer,
Then define two pointer 'h' and 'f', to operate the 'head_t' and 'field_t'.
It is very convenient.

```
     Pointer h       Pointer f       f+1       f+....          f+N-1
|               |              |          |                |              |
-------------------------------------------------------------------------
|    header    |    field1     |  field2 |  ..............  |    field N   |
-------------------------------------------------------------------------
```

- **Why this datastructure have been used?**

If we separate them, then it will be very easy task for us to operate them.
For example, pointer 'f' points to the first field, then we want to set or get data of field2, we just need to increment it i.e. f++, then 'f' will points towards second field.

3. Algorithm detail

Implementation of Bellman-Ford equation :

As we know the Bellman-Ford equations is Dx(y) = minv{c(x,v) + Dv(y)}.

- Important steps in order to implement the successful algorithm :

  Step 1) If **x** equals **y**, the distance is always zero;

  Step 2) Initialize the minimum distance to the cost from **x** to **y**, if **x** and **y** are not neighbor, the cost will be infinite;

  Step 3) travel all the servers, if it is neighbor of source server **x** and the path is not disabled, ignore this server, go to next one;

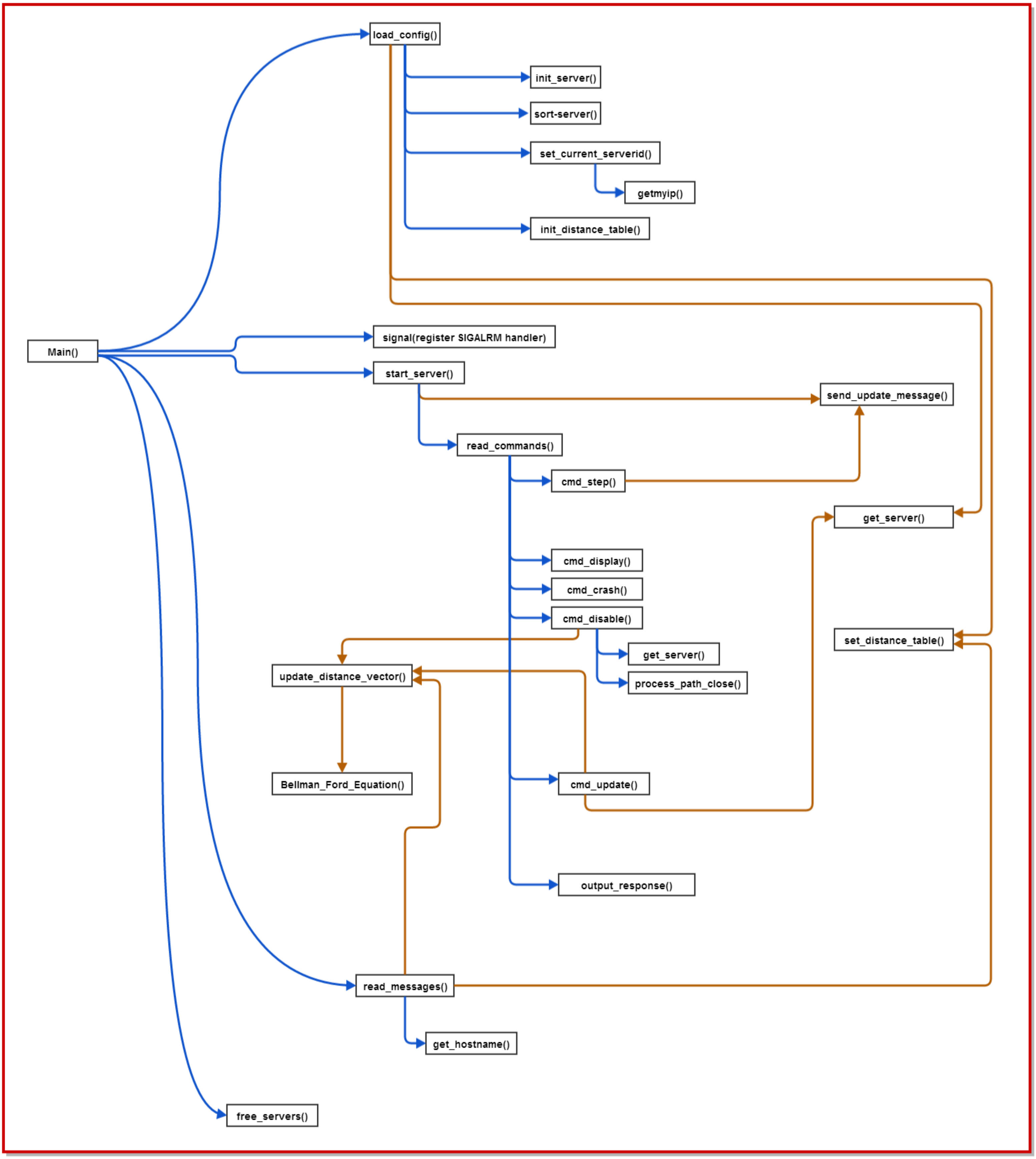
  Step 4) compare d=Dv(y)+c(x,v) with the minimum distance, if **d** is smaller, choose this one as the new minimum distance, and set the forward id to the server **v**.

  Step 5) finally we get the minimum distance, store the back to the distance table.

```
/* Bellman-Ford equation: Dx(y) = minv{c(x,v) + Dv(y)}
        input: y (index of destination server, not server id */
void Bellman_Ford_Equation(int y)
{
        int v, x;
        int d, min, forward;
        struct server *sy;

        /* init start server index, cost and forward */
        x = get_server_index(cur_serverid);
        /* if source and destination server are the same, just return */
        if(x == y) {
                cur_server->cost = 0;
                cur_server->forwardid = cur_serverid;
                distance_table[x][x] = 0;
                return;
        }

        /* set the init distance to the cost directly from x to y */
        sy = &(servers[y]);
        min = sy->cost;
```

```
	forward = y;

	/* calculate all neighbors of current server */
	for(v = 0; v < num_servers; ++v) {
		struct server* sv = &(servers[v]);
		/* if is not neighbor or path closed, go to next server */
		if(!sv->isneighbor)
			continue;
		if(sv->pathclose)
			continue;
		/* calculate the min path */
		d = sv->cost + distance_table[v][y];
		if(d < min) {
			min = d;
			forward = map_ids[v];
		}
	}

	distance_table[x][y] = min;
	servers[y].forwardid = forward;
}
```

4. Function Call Graph & working



Routing Protocol (Distance Vector)

1. **load_config()**

- Open the config file (Topology file);
- Read the first line number of servers;
- Read the second line number of neighbors;
- Allocate memory and init them for all servers;
- Read in server id, IP, port for each server;
- Sort all servers by server-id from small to big;
- Find out the server id of current server, by comparing the IP address;
- Init distance table;
- Read all neighbors of the server;
- Close the config file.

2. **start_server()**

- create a new socket;
- set server socket address;
- bind the socket to the address;
- register a signal handler for SIGINT, because we need to release the socket after exit;
- send route infomation to all neighbors;
- set an alarm clock after time_interval seconds;
- run a forever loop try to listen input from network or user input;

3. **read_messages()**

- Read the packet from the listened port;
- Check the packet size, it must be bigger than the size of struct head_t;
- Read the fields number from header, and check the size of all fields;
- Compare the senders IP with all neighbors, if it is not a neighbor of current server, just drop the packet;
- Get the host name of the sender server;
- If the path from that server is already closed(for example crash, disable), drop the packet;
- Update the last active time of the sender to current time;
- Read all distance from the fields;
- Recalculate the distance vector again.

4. **send_update_message()**

- Fill the IP, port, number of fields to the header;
- Set IP, port, cost, id for all fields;
- Travel through all server, if we are neighbor and the path is not closed, send out message to that server.

5. **read_commands()**

- Read the data from standard input;
- If this server already crashed, do nothing;
- Remove the last new line of input buffer;
- Distinguish the commands, and call different functions to process them;
    - call cmd_step for command step;
    - call cmd_packets for command packets;
    - call cmd_display for command display;
    - call cmd_crash for command crash;
    - call cmd_disable for command disable;
    - call cmd_update for command update;
- If the inserted commands do not match, output an error hint;
- If the command execute successfully, output "command SUCCESS".

6. **cmd_step()**

- Just call function send_update_message, to send out route table to all neighbors.

7. **cmd_packets()**

- Display current packets that we received;
- Clear the packets count to zero.

8. **cmd_crash()**

- Close the path to all other servers, then we will not be receiving or sending any packets anymore;
- Set a crash flag, then we will not be processing any input commands.

9. **cmd_disable()**

- Read the first parameter, which is the destination server id;
- If the destination server id does not exist, return an error;
- If the destination server is not a neighbor, return an error;
- Set the path to closed;
- Recalculate the distance vector;

10. **cmd_update()**

- Read the three parameters id1 id2 and cost;
- If the input cost is a string "inf", set the cost to an integer COST_INFINITE;
- The input cost must between 0 to COST_INFINITE

- If server id1 or id2 does not exist, return an error;
- If id1 equals to id2, return an error;
- We must make sure there is only one 'id' is the id of current server;
- If the destination server is not a neighbor, return an error;
- If the path already closed, return an error;
- Update the cost;
- Recalculate the distance vector.

11. update_distance_vector()

- Travel through all servers, call function Bellman_Ford_Equation to calculate the distance for each server.

12. Bellman_Ford_Equation()

- If the destination 'id' is just current server, set the cost to zero and return;
- Set the minimum cost to the direct cost from current server to the destination server;
- Travel through all servers, which are neighbors and the path is not closed, calculate the cost from that server, if it is less than the minimum cost, reset the minimum cost and set the next hop server to that server;
- Update the distance table and forward id.

5. Observation & Analysis of routing loop

The observation of routing table is based on three servers: sever-1, server-2 and server-3 because if we take 5 servers then it will not be an easy task to observe the flow as there will be so many packets and updates.

Topology:

1<---->2 cost:4
1<---->3 cost:50
2<---->3 cost:1

Network topology :

```
      1
      | \(50)
  (4) |    3
      | /(1)
      2
```

Topology of timberlake server , server-1 :

3
2
1 128.205.36.8 1111
2 128.205.35.24 1112
3 128.205.36.24 1113
1 2 4
1 3 50

Topology of Nickelback server , server-1 :

3
2
1 128.205.36.8 1111
2 128.205.35.24 1112
3 128.205.36.24 1113
2 1 4
2 3 1

Routing Protocol (Distance Vector)

Topology of Metallica server , server-1 :

3
2
1 128.205.36.8 1111
2 128.205.35.24 1112
3 128.205.36.24 1113
3 1 50
3 2 1

Step 1 :

a) Init distance table in server 1:

Table for Server<1><timberlake.cse.buffalo.edu>
1 1 0
2 1 4
3 2 5
DEBUG: Distance Table for Server<1><timberlake.cse.buffalo.edu>:

	1	2	3
1 |	0	4	5
2 |	4	0	1
3 |	5	1	0

DEBUG: Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 4 50

b) Init distance table in server 2:

Table for Server<2><nickelback.cse.buffalo.edu>
1 0 4
2 2 0
3 2 1
DEBUG: Distance Table for Server<2><nickelback.cse.buffalo.edu>:

	1	2	3
1 |	0	4	5
2 |	4	0	1
3 |	5	1	0

DEBUG: Cost Table for Server<2><nickelback.cse.buffalo.edu>: 4 0 1

c) Init distance table in server 3:

Table for Server<3><metallica.cse.buffalo.edu>
1 2 5
2 1 1
3 3 0

```
DEBUG: Distance Table for Server<3><metallica.cse.buffalo.edu> :
         1    2    3
       ------------
  1 |    0    4    5
  2 |    4    0    1
  3 |    5    1    0
DEBUG: Cost Table for Server<3><metallica.cse.buffalo.edu> : 50 1 0
```

Step 2 :
This time "update 2 3 inf" in server 2 and "update 3 2 inf" in server 3.

a) Distance table in server 3:

```
Table for Server<3><metallica.cse.buffalo.edu>
1 0 50
2 1 54
3 3 0
DEBUG: Distance Table for Server<3><metallica.cse.buffalo.edu> :
         1    2    3
       ------------
  1 |    0    4    5
  2 |    4    0    1
  3 | 50    54    0
DEBUG: Cost Table for Server<3><metallica.cse.buffalo.edu> : 50 - 0
```

Observation :

From the observation, we can find the distance from the server 3 to 2 changed to 54 immediately. **Why?**

Analysis :

According to the Distance Vector algorithm, we can calculate distance from server-3 to server-2, but to answer the above scenario there can be two possibilities.

i) One way server-3 directly go to server-2;
ii) The other way server-3 go to server-2 through server-1.

Thus calculating distance :

D3(2) = min{c(3,2)+d(2,2), c(3,1)+d(1,2)} = min{inf, 50+4} = 54

b) Distance table in server 2:

Iteration 1 :

Table for Server<2><nickelback.cse.buffalo.edu>
1 0 4
2 2 0
3 1 9
DEBUG: Distance Table for Server<2><nickelback.cse.buffalo.edu>:

	1	2	3
1 |	0	4	5
2 |	4	0	9
3 |	5	1	0

DEBUG: Cost Table for Server<2><nickelback.cse.buffalo.edu> : 4 0 -

Iteration 2 :

Table for Server<2><nickelback.cse.buffalo.edu>
1 0 4
2 2 0
3 1 17
DEBUG: Distance Table for Server<2><nickelback.cse.buffalo.edu>:

	1	2	3
1 |	0	4	13
2 |	4	0	17
3 |	50	54	0

DEBUG: Cost Table for Server<2><nickelback.cse.buffalo.edu>: 4 0 -

Iteration 3 :

Table for Server<2><nickelback.cse.buffalo.edu>
1 0 4
2 2 0
3 1 49
DEBUG: Distance Table for Server<2><nickelback.cse.buffalo.edu>:

	1	2	3
1 |	0	4	45
2 |	4	0	49
3 |	50	54	0

DEBUG: Cost Table for Server<2><nickelback.cse.buffalo.edu>: 4 0 -

<u>Iteration 4</u> :

```
Table for Server<2><nickelback.cse.buffalo.edu>
1 0 4
2 2 0
3 1 54
DEBUG: Distance Table for Server<2><nickelback.cse.buffalo.edu>:
        1    2    3
       -------------
 1 |    0    4 50
 2 |    4    0 54
 3 | 50 54    0
DEBUG: Cost Table for Server<2><nickelback.cse.buffalo.edu>: 4 0 -
```

c) Distance table in server 1:

<u>Iteration 1</u> :

```
Table for Server<1><timberlake.cse.buffalo.edu>
1 1 0
2 1 4
3 2 13
DEBUG: Distance Table for Server<1><timberlake.cse.buffalo.edu>:
        1    2    3
       ------------
 1 |    0    4 13
 2 |    4    0    9
 3 | 50 54    0
DEBUG: Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 4 50
```

<u>Iteration 2</u> :

```
Table for Server<1><timberlake.cse.buffalo.edu>
1 1 0
2 1 4
3 2 21
DEBUG: Distance Table for Server<1><timberlake.cse.buffalo.edu>:
        1    2    3
       ------------
 1 |    0    4 21
 2 |    4    0 17
 3 | 50 54    0
DEBUG: Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 4 50
```

Iteration 3 :

```
Table for Server<1><timberlake.cse.buffalo.edu>
1 1 0
2 1 4
3 2 50
DEBUG: Distance Table for Server<1><timberlake.cse.buffalo.edu>:
        1    2    3
      ------------
 1 |    0    4 50
 2 |    4    0 49
 3 | 50 54    0
DEBUG: Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 4 50
```

Iteration 4 :

```
Table for Server<1><timberlake.cse.buffalo.edu>
1 1 0
2 1 4
3 2 50
DEBUG: Distance Table for Server<1><timberlake.cse.buffalo.edu>:
        1    2    3
      ------------
 1 |    0    4 50
 2 |    4    0 54
 3 | 50 54    0
DEBUG: Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 4 50
```

Observation :

From the observation, why the distance do not change immediately?

Analysis :

As we know, after we update the cost, the correct distance from server-2 to server-3 should be 54, from server-1 to server-3 should be 50 but from the observation we can see the distance did not change.

In server-2:

Before the command "update", our D1(3)=5

After the command:

D2(3) = min(c(2,3)+d(3,3), c(2,1)+d(1,3)} = min{inf, 4+5} = 9

Then we send out distance (D2(3)=9) to server-1:

In server-1: (receive D2(3)=9)
D1(3) = min{c(1,3)+d(3,3), c(1,2)+d(2,3)} = min{50+0, 4+9} = 13
Then we send out distance (D1(3)=13) to server-2:

In server-2: (receive D1(3)=13)
D2(3) = min(c(2,3)+d(3,3), c(2,1)+d(1,3)} = min{inf, 4+13} = 17
Then we send out distance (D2(3)=17) to server-1:

In server-1: (receive D2(3)=17)
D1(3) = min{c(1,3)+d(3,3), c(1,2)+d(2,3)} = min{50+0, 4+17} = 21
Then we send out distance (D1(3)=21) to server-2:

In server-2: (recieve D1(3)=21)
D2(3) = min(c(2,3)+d(3,3), c(2,1)+d(1,3)} = min{inf, 4+21} = 25
Then we send out distance (D2(3)=25) to server-1:

and so on...

- From the calculation we can see the distance will be correct after many rounds of exchange messages.
- The cost adjustment transfer process is very slow that is why server-1 and server-2 did not change the cost immediately.

6. Project Snapshots

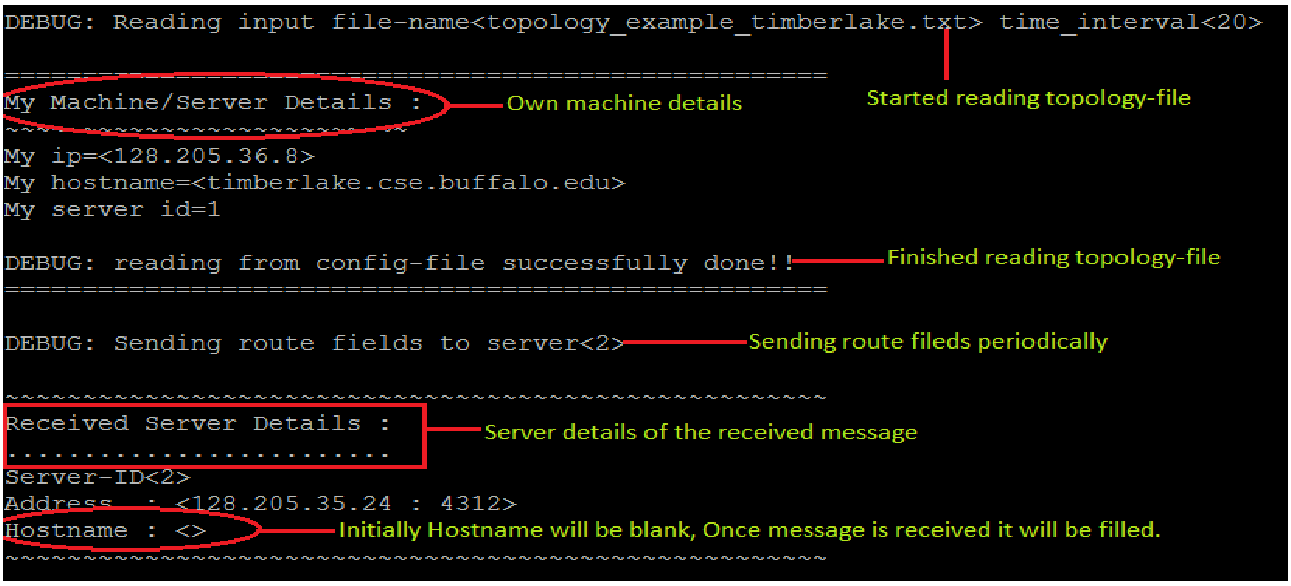
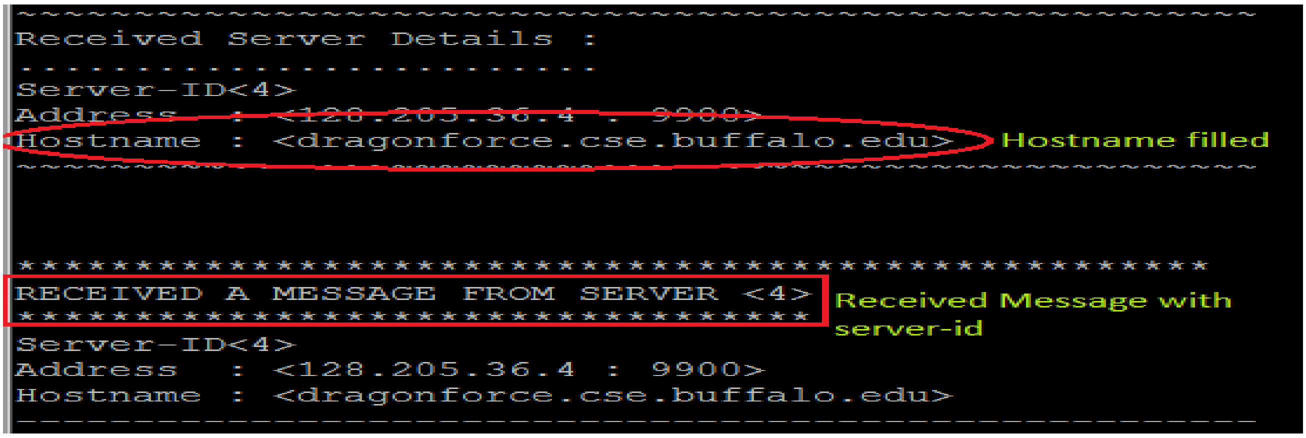


Fig (a)

Initially the hostname field will be blank so once the server received the message from its neighbor, it will automatically retrieve the received server credentials and display it into the respective fields.

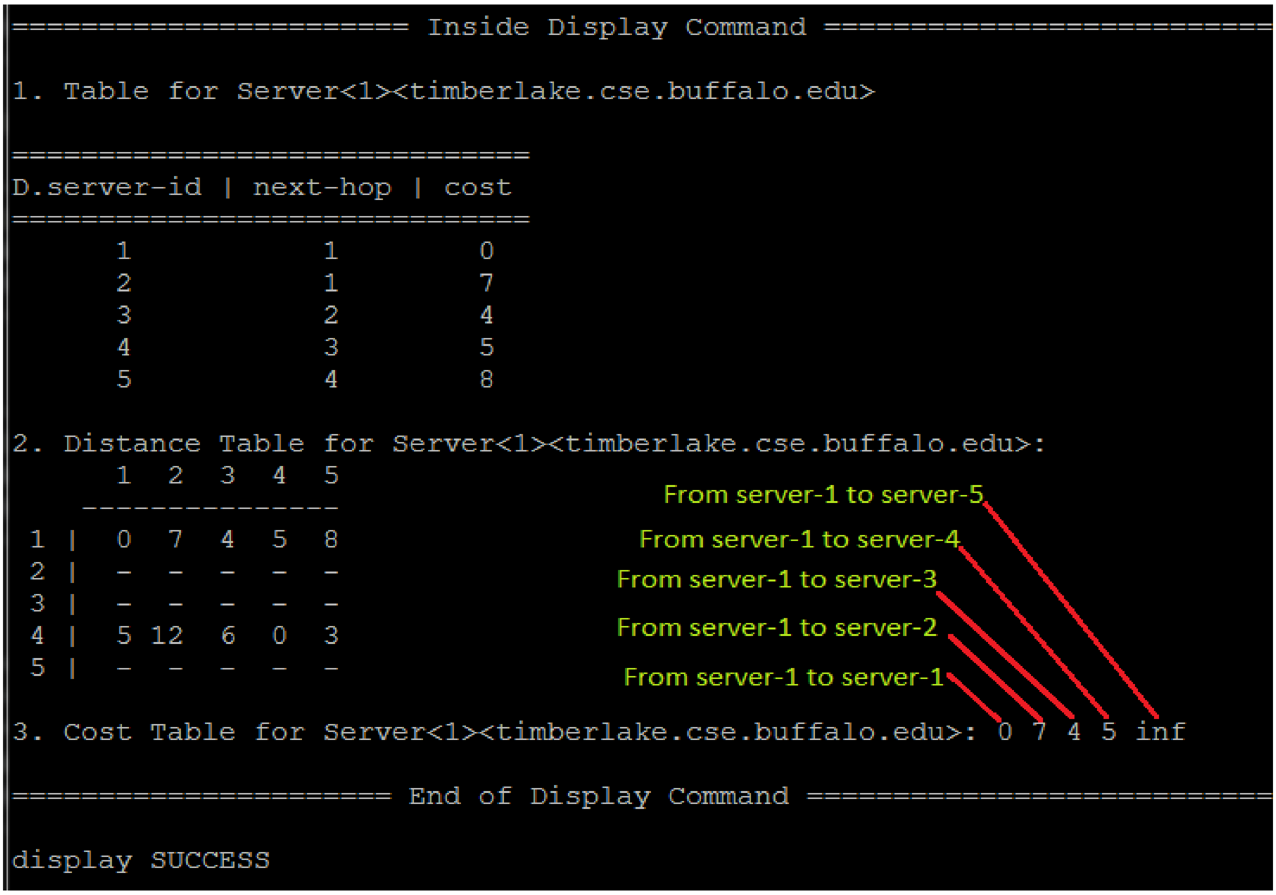
This (above) process will run once.



Fig(b)

This is the output of received message, now the receiving server retrieved the hostname, sever-id, port number, IP of the sending/neighbors server.

We are retrieving these details based on public DNS which have been crated, In this the IP address should be the public IP address of the system which will be then found by connecting created public DNS server.




```
====================== Inside Display Command ==========================

1. Table for Server<1><timberlake.cse.buffalo.edu>

==============================
D.server-id | next-hop | cost
==============================
      1            1        0
      2            1        7
      3            2        4
      4            3        5
      5            4        8

2. Distance Table for Server<1><timberlake.cse.buffalo.edu>:
      1  2  3  4  5
    ---------------
 1 |  0  7  4  5  8
 2 |  -  -  -  -  -
 3 |  -  -  -  -  -
 4 |  5 12  6  0  3
 5 |  -  -  -  -  -

3. Cost Table for Server<1><timberlake.cse.buffalo.edu>: 0 7 4 5 inf

====================== End of Display Command ==========================

display SUCCESS
```

Fig(c)

The "display" command will display 3 tables :

i) Server Table which defines 3 fields.
   a. Destination server-id
   b. Next hop
   c. Path cost

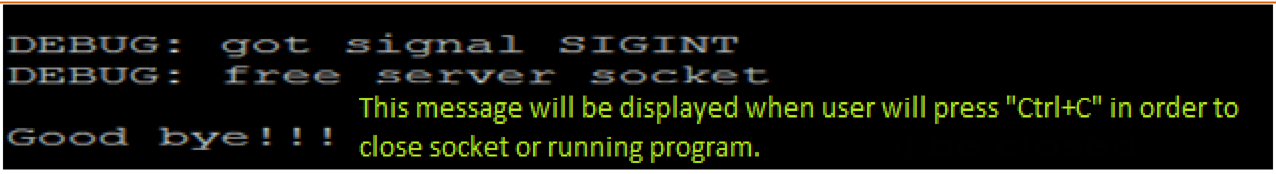
ii) Distance Table

iii) Cost Table

The cost table can be consider as : (from snapshot)

i)	0	:	From server-1 to server-1 cost is "0"
ii)	7	:	From server-1 to server-2 cost is "7"
iii)	4	:	From server-1 to server-3 cost is "4"
iv)	5	:	From server-1 to server-4 cost is "5"
v)	Inf	:	From server-1 to server-5 cost is "inf"

Santosh Kumar Dubey
Person # 5009 7059



Fig(d)

Now, this function will be called when the user will press “Ctrl+C” button in order to close/stop running program. This will be called to free running socket/port.

7. References

- http://en.wikipedia.org/wiki/Distance-vector_routing_protocol
- http://www.inetdaemon.com/tutorials/internet/ip/routing/dv_vs_ls.shtml
- http://www.ciscopress.com/articles/article.asp?p=24090&seqNum=3
- http://www.firewall.cx/networking-topics/routing/routing-protocols/182-distance-vector.html
- http://sourcecode4all.wordpress.com/2012/04/01/distance-vector-routing-algorithm-in-c/
- http://www.slideshare.net/ergauravrawat/distance-vector-routing-algorithm
- http://www.csanimated.com/animation.php?t=Bellman-Ford_algorithm
- http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/