Networking Assignment 3
Distance Vector
By Omkar Bhide & Snehil Vishwakarma

Data Structures

We have used two dimensional vectors and structures to store graph and routing tables. Structure graph contains two nodes and weight variables which denote the graph, as follows:

```
struct graph
{
char node_r,node_c;
int weight;
};
```

Instance of 'graph'

```
std::vector<std::vector<graph> > v2d;
```

Structure 'routing' stores the routing table, with destination node, next node, cost, ttl as its variables. It is as follows:

```
struct routing
{
   char dest_node,next_node;
   int cost;
   unsigned short ttl;
};
```

Instances of 'routing'

```
std::vector<routing> rt;          ..stores routing table
std::vector<routing> rt_recv;     ..stores received routing tables temporarily
```

Multithreading

We have implemented multithreading, using two separate threads, one for sending advertisements (send thread) and another for receiving and processing the advertisements (update thread)

Shared Variables:
'rt_recv' is the only shared variable between the two threads.

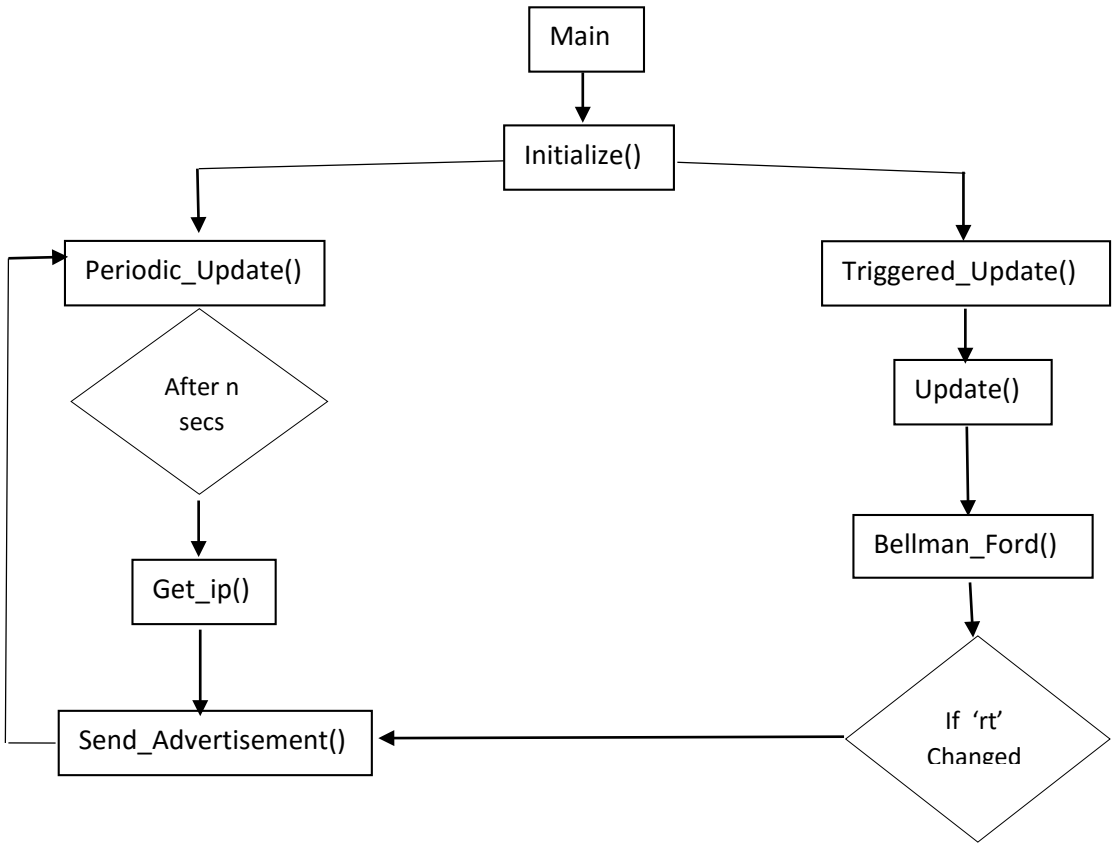
Mutex and Conditional Variables:
We have used mutex for '*std::cout*'. Though we thought of using mutex and conditional variables for '*rt*' routing table, we didn't felt the need, since our code is running fine, without any data races.

Program Modules and Flow

Methods:

Main – handles arguments, initialize and threads
Initialize – Initializes the graph, routing table from configuration file
Periodic_update – Thread that periodically sends advertisements
Triggered_update – Thread that implements triggered updates
Send_Advertisement – Sends routing table to neighbours
Update –Processes received advertisements
Bellman_ford – Updates the routing table
Get_ip – Used for mapping from node to ip

Flow:



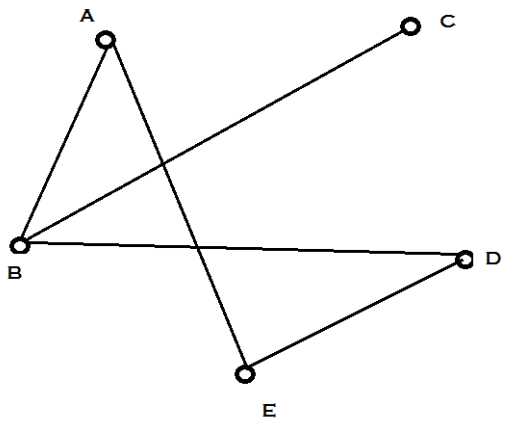
Assumptions:

1. Our code supports up-to 26 nodes (*a-z*).
2. Works only for non-weighted and positive weighted graphs.
3. Node naming must start from '*a*' and continue sequentially.
4. Command line arguments are as follows:
   a. ./dv.out ..program name
   b. config# ..configuration file for the node
   c. port_no ..unique port number
   d. ttl ..time to live
   e. infinity ..max hop count
   f. period ..periodic update interval
   g. split_horizon ..0 or 1

5. Configuration file should have nodes in ascending order, excluding the own node. For eg. '*configa*' would look like:
   b yes
   c no
6. Compiling:
   g++ -pthread dv.cpp –o dv.out

Time Analysis

Sample graph (parameters: ttl 90, infinity 16, period 15)



Outputs (Stable Routing Tables):

A

```
a a 0 90
b b 1 90
c b 2 90
d b 2 90
e e 1 90
```

B

```
a a 1 90
b b 0 90
c c 1 90
d d 1 90
e a 2 90
```

C

```
a b 2 90
b b 1 90
c c 0 90
d b 2 90
e b 3 90
```

D

```
a e 2 90
b b 1 90
c b 2 90
d d 0 90
e e 1 90
```

E

```
a a 1 90
b d 2 90
c a 3 90
d d 1 75
e e 0 90
```

1. Time required to establish routes to all nodes
   Convergence for the above graph with 5 nodes and ttl 90, infinity 16 and period 15 was **30 secs** approx.

2. Converge to steady state after a node goes down
   When node E was simulated to go down, it took about **x secs** for all the routing tables to update and come to a stable state.

3. Effect of different infinity (max hop) values on time required for convergence
   Using infinity as 4 was more effective (faster) than using infinity of 16.

4. Effect of split horizon on the time required for convergence.
   The above graph with ttl 90, infinity 16 and period 15 came to a stable state in almost **16 secs** using split horizon. Which is faster than without split horizon.

Note: The timings stated were manually observed using stop watch, and thus includes human interpretation delays.