CSE 589 PROJECT 2

CODE ORGANIZATION

&

REPORT

CODE DETAILS:

The Program is split into multiple ".c" and ".h" files:

All source code is organized into two folders, namely: *"src"* and *"include"* :

1. **src** : has all the .c files source files
2. **include**: has all the .h files

There is a makefile named as *"makefile"* which shall create an executable of name **"server"**

Definiton of INFINITY – 65535

Since "COST" is supposed to be 16 bits long and positive therefore INIFNITY is nothing but the maximum value that can be held by an unsigned short int viz. 65535

Files inside "src" folder: (All functions listed below have comments in their source file)

1. **CommandOperation.c** – Has function that enable the application to parse User Input (Console Input)
    a. int commandMaster(char* command)
2. **rajaramr_proj2.c** – The main program from where all things good start
    a. int main(int argc, char** argv);
    b. void usage(char* applicationName);
    c. void getThisMachineIP();
    d. void printNetworkInfo(networkInfo * network);
3. **serverOps.c** – Here lies the engine of the application from maintaining Communication, DV calculation to what not

    FUNCTIONS THAT KEEP THE SERVER RUNNING

    a. int getServerID(char* serverIP,networkInfo* network);
    b. char* constructMessagePacket(networkInfo* network,int* messageSize,unsigned short int** costMatrix);
    c. int parseMessage_n_updateCostMatrix(char* message,int messageSize,networkInfo* network,int** nextHop,routingTableRecord** rt,int myServerID,int senderServerID,unsigned short int*** costMatrix);
    d. unsigned short int** createCostMatrix(unsigned short int nodes,int neighbourCount,links* edges);
    e. int distanceVector_Calc(int senderServerID,neighbour* neighbourList,int** nextHop,networkInfo** network,unsigned short int*** matrix);
    f. void sendData_to_Neighbours(int sockFD,neighbour* neighbourList,int neighbourCount,char *message,int messageSize);
    g. int isNeighbour(neighbour* neighbourList,int neighbourCount,unsigned short int targetServerID);

FUNCTIONS THAT HANDLE USER INPUT

a. void packets(int *pkts);
b. int update(char** commandArgs,networkInfo** network,int** nextHop,unsigned short int*** costMatrix,neighbour* neighbourList);
c. int disable(int** nextHop,neighbour *neighbourList,networkInfo** network,unsigned short int*** costMatrix);

4. **routingTableOps.c** -- Has functions that enable the application to create/manage routing tables
    a. routingTableRecord* contructRoutingTable(networkInfo * network);
    b. int printRoutingTable(routingTableRecord * rec,int nodes);
    c. int updateRoutingTable(int myServerID,int* nextHop,int nodes,routingTableRecord** routingTable,unsigned short int** costMatrix);
5. **topologyFileReader.c** -- Has functions to read a topology file and make sense of the same + commit info gained into a struct
    a. networkInfo* getTopologyInfo(char* fileName);
    b. int parseTopologyData(networkInfo** network, char* topoData)

B. Files inside include folder:

The "include" folder only contains .h files, the following:

1. ***appMacros.h*** – Application wide macros are declared here
2. ***commandOperation.h*** – Exposes the "commandMaster" function of the corresponding .C file
3. ***error.h*** – Has the error macros for the Application -- e.g. SUCCESS, FAILURE, TRUE and FALSE
4. ***globalVars.h*** – A convenient way to share global variables declared in "rajaramr_porj2.c"
5. ***messageTemplate.h*** – Has the definition of the struct to be used for Message Construction AND Parsing
6. ***neighbourDetails.h*** – Has the definition of the struct that shall be used to store the neighbour level info
7. ***routingTableOps.h*** – Exposes the functions to create AND maintain a routingTable
8. ***serverOps.h*** – Exposes the "serverStartup" function to outside world
9. ***topologyFileReader.h*** – Exposes the functions required to Read a topology file

C. Structures Used

The Structure for the message format: Can be found in messageTemplate.h

```
typedef struct messageNugget --- (Line Number 19)
{
        char serverIP[INET_ADDRSTRLEN];
        short int serverPort;
        short int serverID;
        unsigned short int cost;
}msgNugget;
typedef struct messageFormat --- (Line Number 33)
{
        short int numUpdateFields;
        short int serverPort;
        char serverIP[INET_ADDRSTRLEN];
}msgFormat;
```

This structure is modified and used in the following functions of **serverOps.c** –

- char* constructMessagePacket(networkInfo* network,int* messageSize,unsigned short int** costMatrix);
- int parseMessage_n_updateCostMatrix(char* message,int messageSize,networkInfo* network,int** nextHop,routingTableRecord** rt,int myServerID,int senderServerID,unsigned short int*** costMatrix);

The structure for Routing table: Can be found in routingTableTemplate.h

```
typedef struct routingTableRecord -- (Line Number 13)
{
        int destServerID;
        int nextHopServerID;
        unsigned short int costOfPath;
}routingTableRecord;
```

It is used and manipulated in the following functions:

- routingTableRecord* contructRoutingTable(networkInfo * network);
- int printRoutingTable(routingTableRecord * rec,int nodes);
- int updateRoutingTable(int myServerID,int* nextHop,int nodes,routingTableRecord** routingTable,unsigned short int** costMatrix);

Structure to store network details:

In file "neighbourDetails.h": Used to store neighbor's information

```
typedef struct neighbour --- (Line Number 18)
{
        short int serverID;
        char ipAddress[INET_ADDRSTRLEN];
        int port; // fixme
        int ignore;
        int timeoutCount;
        int disAllowUpdate;
        unsigned short int cost;
        struct sockaddr_in neighbourAddress;
}neighbour;
```

in the "serverDetails.h" :Used to store details of servers on the network

```
// Information about a server on the network -- child struct

typedef struct serverInfo (Line Number 24)
{
        short int serverID;
        char serverIP[INET_ADDRSTRLEN];
        short int serverPort;
        unsigned short int cost;
        int isNeighbour; // true or false
        struct serverInfo *next;
}serverInfo;


// the adjacent edges and the associated cost -- child struct
typedef struct links (Line Number 34)
{
        int startNode;
        int endNode;
        int linkCost;
}links;

// Shall be used to hold all the information about the network
// + information about self
typedef struct networkInfo (Line Number 35)
{
        short int serverCount;
        short int neighbourCount;
        short int myServerID;
        short int myPort;
        char myIP[INET_ADDRSTRLEN];
        serverInfo* serversOnNetwork;
```

```
        links* edges;
}networkInfo;
```

D. Cost Matrix to calculate distances

I also use a cost -matrix which is filled in with the details from the servers own adjacent edges, distance vectors of neighbor:
The function used to create the cost-matrix; All these functions are in "**serverOps.c**":

1. **Create Matrix :** unsigned short int** createCostMatrix(unsigned short int nodes,int neighbourCount,links* edges);
2. **Parse Message :** int parseMessage_n_updateCostMatrix(char* message,int messageSize,networkInfo* network,int** nextHop,routingTableRecord** rt,int myServerID,int senderServerID,unsigned short int*** costMatrix);
3. **Construct Message** : char* constructMessagePacket(networkInfo* network,int* messageSize,unsigned short int** costMatrix);

REPORT

The distance vector is calculated using the Bellman-Ford algorithm. The functions that are used to maintain the sanity of the path and routes are coded in the "**serverOps.c**" source file, and they are:

- parseMessage_n_updateCostMatrix(params) – Prases neighbour's message and updates cost-matrix
- distanceVector_Calc(params) – Implements Bellman – Ford and updates cost-matrix
- disable(params) – Implements "disable" command and updates cost-matrix
- update(params) – Implements "update" command and updates cost-matrix

When the program is run on different servers in unison:

- The costs from One server to others (Neighbours and their Neighbours) is calculated using the distance vectors received from neighbours
- A server is aware of the existence of its neighbours only and is oblivious about other servers on the network to start with and then learns about other servers and the distances to them from the neighbours' messages

With the passage of time a server learns and figures out path to its neighbours' neighbours
And the routing tables of all the servers converge and stabilize

When links are updated – Late convergence

When links are updated then it takes a while for a neighbor to reconcile to the new costs as it does not know about the update until a neighbor tells it about the modifications that have been made to its adjacent edges. Due to the very nature of the system (updates being sent at intervals rather than when a modification event is encountered) the program goes into a destabilized state wherein

- A server shall mislead its neighbor about the cost of a path that goes through the very same neighbor

- Leading to a case wherein the algorithms shall compute wrong routes and bad costs to a neighbour's neighbor (1 degree of separation or more)
- This shall result in the routing tables showing wrong/corrupted information – till such a time when the algorithms learns that there is a mistake and corrects itself leading to a convergence
- This bad feeback shall continue till convergence is achieved and the time taken for the convergence to be achieved is directly proportional to the cost update value which made the system go into a bad state in the first place

Infinity by Induction

The program has been designed to handle all eventualities described in the project description:

- When a server reached another server by hopping through its neighbor, if the neighbor reports that it cannot reach the destination that previously reachable, in its distance vector then the cost to that destination is set to "INFINITY" and the next hope is set to "NONE"

I would like to call this behavior of the code as infinity by induction.

However if the destination is indeed a neighbor of the server running the code then the destination is still reachable directly thanks to a direct physical link as reported by the topology file. The program makes accommodation for all this and computes the right cost and next hop values and advertises the same to its neighbours.

This behavior of the program helps the Cost and the Distance vectors to converge after being misled for a few cycles

Timeout Induced Infinity

When a neighbor times-out (no communication for a period == 3 * Update_intervals), it is only logical that paths being routed through the server that has timed out be set to INFINITY and the next hop for those paths be set to "NONE" until the some other neighbor comes back with information about a new route to the server that has timeout and other servers that were previously routed through the neighbor that had just gone done

Count to Infinity

When a server has only one edge and all other servers used that path to communicate to the former, a disabling of link/crash of the former server can lead to some bad and bizarre consequences. The key here is that the information that a server has gone done has to be propagated very quickly across the network by its, in this case, neighbor (Just one neighbor here because this is a special case). If the message is not propagated to all the servers in the network in time it shall result in a condition where one server misleads the other server about the existence of a non-existing path thereby resulting in a count to infinity problem

This happens primarily due to the servers not sharing their next hop information along with the cost messages with their adjacent servers.

The program goes into a downward spiral and never recovers due to lack of servers who can inject sanity into the network by feeding right information.