3D3 Computer Networks Project 2

Group 1
Nalin Kamboj - 17317423
Xuming Xiu - 14332504
Peter Rymsza - 17344435
Conor Quinn - 14318836

Implementation Description

Data Structures -
The structures routerData, nRouter (short for neighbor router), and destinationRouter have very similar structures, but we created those for making our code easier to understand and expand as we added new features. We have also used a few struct objects as global variables (which is not always a good idea), but we did it anyway to further simplify our implementation. This helped us reduce the number of arguments passed around between various functions.

```
struct nRouter{     //Struct
    char src;
    int port;
    int weight;
}neighbors[MAX_ROUTERS];
```

Structures edge, graph, and DV are used for storing information relevant to our routing table and maintaining our graph of active routers and edges in the network.

```
struct graph {
    int V, E;
    vector<struct edge*> edges;
};
```

Finally, structure discoveredRouters (similar to our other router structures as well) is used to store router info which are ‘not’ neighbors. This helped us easily distinguish between different routers and simplify implementation of our routing algorithm.

Packet Structure -
One of the most important decisions we made during the project was deciding an efficient and easy (format?) (what?)to implement packet structure.

Our finalized packet structure -
DESTN:<DESTN>,SRC:<SRC>,FUNC:<FUNC>,TYPE:1/0,PORT:<PORT>,MSG:<MSG>,<ACTUALDESTN>

The last field <ACTUALDESTN> wasn’t a part of our initial packet structure, but as we progressed, we realized it was necessary for us to store the router from which the packet was “originally” transmitted. This is necessary for us to figure out the packet’s optimal path and not lose track of its origin.

<FUNC> denotes what type of data/command the packet is carrying for particular router. Each <FUNC> denotes a particular function to be called by the router -
1 - This again has two parts. The field <TYPE> denotes whether a router is requesting for router information or sending router information.
3 - Transmit message

Functions/Threads -
The two threads “connection_thread” and “sender_thread” are responsible for handling data receiving and user input respectively. While the name “sender_thread” suggests that this thread is used for sending data, it is actually used to take user input and call a function which appropriately sends data relevant to the choice the user entered.

```
int ch;
cout<<"\n--------------------------------------------------------------------------------";
cout<<"\nMESSENGER: 1 - Initialize Nodes, 2 - Shutdown, 3 - View Routing Table\n\t Enter choice: ";
cin>>ch;
```

“packandsend” and “packetParser” are probably the most key functions in the router program. While “packandsend” is responsible for generating the appropriate packet (includes generating the header, attaching the user’s message etc) and sending it to the proper router (by looking up the routing table), “packetParser” is responsible for handling every packet received by the router. packParser decodes the header information and is responsible for deciding whether or not to reply, and what to reply to any router. “packandsend” can also take up at most 7 arguments, which is a lot for any function. For this reason, 5 of the arguments are provided with a default value, and these arguments are only required for very specific packet types.

```
void packandsend(struct destinationRouter *destnR, int funcType, int type = 1, string message = "", struct edge *ed = NULL,
int storeorsend = 0, char actualDestination = -1) {
    int sendLen = sizeof(struct sockaddr_in);
```

“insertEdge” governs the insertion of edges into the graph while making sure it does not insert an edge which already exists. As mentioned earlier, this function is called by “packetParser” when it detects that a router has sent edge information.

Finally, “BellmanFord” function is responsible for actually applying the Bellman Ford algorithm to the discovered routers and generating the routing table, which it prints out to a file using writeDVinfo() function.

Program Flow

As a first step, main() initializes the neighbors by reading from INIT_FILE, and initializes an empty graph. Next, main() applies Bellman-Ford algorithm to the current graph and finally sets up sender_thread and connection_thread which drive the rest of the program.

In our approach, we did not come up with a way for routers to start their communication automatically as we didn’t think it was that important. Instead, once we are ready and we fire up all our routers, we call function type ‘1’ using the program’s menu driven flow and the router begins to start exchanging DVs with other routers. This step has to be manually carried out on all the other routers, although it might not have any effect. That is because as routers receive packets, the packetParser() takes care of router discovery as well, to a certain extent.

Finally, once initialization is done, we run the algorithm and generate shortest path data for every router on the network and write it out to a file. This step is also carried out at multiple stages during the program to enable concurrent updation of the routing tables.

For the sake of simplicity, we have not included the negative-cycle detection part of the algorithm in our code because in this scenario, we have assumed that the weights between the routers can never be negative. Hence we decided to skip that part altogether, although we do realize the importance of this step in real-world applications.

```
for (int i = 1; i <= V-1; i++)
{
    for (int j = 0; j < E; j++)
    {
        int u = g->edges[j]->v1;
        int v = g->edges[j]->v2;
        int weight = g->edges[j]->weight;
        //if (dist[u-65] != 10000 && dist[u-65] + weight < dist[v-65])
        //    dist[v-65] = dist[u-65] + weight;
        if(dvinfo[u%65].shortestDist != 10000 && dvinfo[u%65].shortestDist + weight < dvinfo[v%65].shortestDist) {
            dvinfo[v%65].shortestDist = dvinfo[u%65].shortestDist + weight;
            dvinfo[v%65].nextNode = dvinfo[u%65].node;
        }
    }
}
```

Other Key Decisions

1) We decided to use <vector> in the data structure graph and for creating objects of discoveredRouters data structure. We decided to use vector because it offers easy to use functions for dynamically expanding its size and appending/adding elements to the vector array easily and efficiently.
2) We had lots of trouble in coming up with an easy way to store the output of the Bellman-Ford algorithm and hence we finally came up with the DV data structure.