CAN201 Introduction to Networking
Coursework 2
Routing simulation

Student Name: Yuxuan.Ren

Student ID: 1823678

Routing simulation

Abstract

Router is an important part of network. One function of a router is routing, getting good paths from sending hosts to receiving hosts through routing algorithms. There are many kinds of routing algorithms. For example, Bellman-Ford distance vector algorithm and Dijkstra algorithm. The purpose of this report is to implement Bellman-Ford distance vector algorithm using Python Socket network programming.

Keywords: routing, Bellman-Ford distance vector algorithm, Python

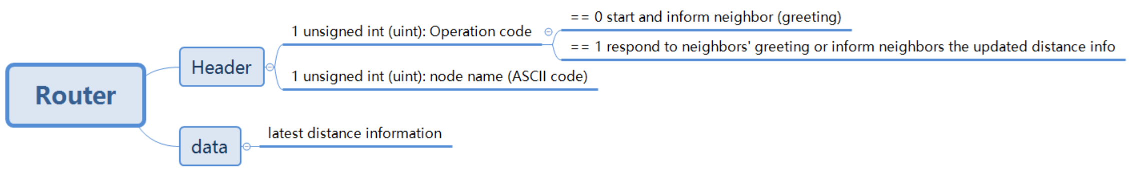
1. Introduction

Router is an essential component of today's network. Without it, the network would be more complexed. The functions of a router are routing and forwarding. The aim of forwarding is to move packets from router's input to appropriate router output and the aim of routing is to determine route taken by packets from source to destination. Routing is a "top 10" networking challenge. All routing algorithms can be classified into two types. One is link state algorithm that each router gets complete info about the topology, the other is distance vector algorithm that each router gets part of the info about the topology. The most well-known routing algorithms are Dijkstra algorithm (link state algorithm) and Bellman-Ford algorithm (distance vector algorithm). There is also another algorithm for wireless networks called temporally-ordered routing algorithm, which performs better for network of large size [1][2]. The aim of this project is to implement Bellman-Ford distance vector algorithm using Python Socket networking programming. The project should be able to get the shortest distances correctly through several updates with the given information and output the final result in a json file if there is no update any more. The remainder of this report is organized as follows. Section two discusses the methodology. Section three shows how the project is really implemented. In section four, a testing plan and its results will be given to show the correctness of the implementation. Finally, the report will end with a conclusion and some constructive suggestions.

2. Methodology

2.1. Proposed protocols

Operation code 0 for greeting and operation code 1 for informing changes or responding to greeting.


Router
Header
1 unsigned int (uint): Operation code
== 0 start and inform neighbor (greeting)
== 1 respond to neighbors' greeting or inform neighbors the updated distance info
1 unsigned int (uint): node name (ASCII code)
data
latest distance information

2.2. Proposed functions

According to the Bellman-Ford distance vector algorithm, each node maintains two information, the direct distances to its neighbors (c(x, y): distance between x and y) and the currently shortest distances to other routers (distance vector or DV).

Each node should have three main functions. 1)Read given information (IP info and direct distances to its neighbors) from json files in the current working directory. 2) Notify neighbors if its DV changes or it is started. 3) Recalculate distance vector. For the third function, the recalculation should base on Bellman-Ford algorithm. When receiving a DV update message from neighbors, the node should update the distance vector in this way: Dx(y) = min {Dx(y), c(x, v)+Dv(y)}. The node should only communicate with its neighbors and no IP information of other nodes should be achieved during running.

2.3. Ideas

1. The communications between nodes should use UDP protocol.
2. Use several global variables to store the information. the local IP address, IP addresses of neighbors, direct distances to neighbors and its distance vector.
3. Use some global variables to store the information whether the DV was updated in the latest recalculation. Then based on the variables to decide whether the DV should be sent to neighbors.
4. When reading files has finished, the distance vector should be written once using the info from files and send it to the neighbors.

5. If the node does not receive new message for a period of time, the current DV will be output into a JSON file and the program will terminate.
6. Since the router will not be killed after it is started, there is no need to send DV to neighbors from time to time.

3. Implementation

3.1. Steps of implementation

The variables are node_name (stores local port number), neighbor_ip (a dictionary that stores neighbor's port info), neighbor_distance (a dictionary that stores direct distances to neighbors), distance_output (a dictionary that stores distance vector together with "next_hop" info) and distance_update (an int that shows whether the DV changes or not).

The implementation is divided into five steps based on the three main functions: 1) Implement reading function. 2) Implement making package function. 3) Implement processing package methods. 4) Implement notifying functions. 5) Decide the terminating condition.

The first step:

After starting the program with node name, the program should know its node name. For my program, it is stored in node_name. Then it can use node_name + '_ip.json' and node_name + '_distance.json' to open and read the related JSON files. For every IP info, if the node name is itself, just use it to bind the socket, otherwise, store it in neighbor_ip. For the distance info, store them in the neighbor_distance and distance_output. The "next_hop" info should just be the distance name.

The second step:

The package should contain an operation code, a node name and the DV info. According to the proposed protocol in 2.1, if the operation code equals to 0, it means that the node is just started and gives the initial info to its neighbors (say hello), the neighbor should also give its DV. If the operation code equals to 1, it means that the node answers to its neighbor's greeting or notifies neighbors of the latest DV. The DV info can be got by just using the node name and distance in distance_output to form a new dictionary.

The third step:

When receiving a message from neighbor. It should first store the operation code (using when sending the message). Then recalculate the DV. There are two situations: 1) The node name in the received DV is not in the local distance_output. If it happens, just adding the node name into distance_output. The distance is the distance to neighbor + the distance from neighbor to that node, the "next_hop" is node name of neighbor. Then, set the distance_update variable to 1 which means that the DV is updated and neighbors need to be notified 2) The node name in the received DV is in the local distance_output. If it happens, just compare the two distances (the original one and the new one). if the original one is smaller, no need to update the DV. If the new one is smaller, update the DV using the new distance. And the "next_hop" is the node name of neighbor. In addition, the distance_update variable should be set to 1.

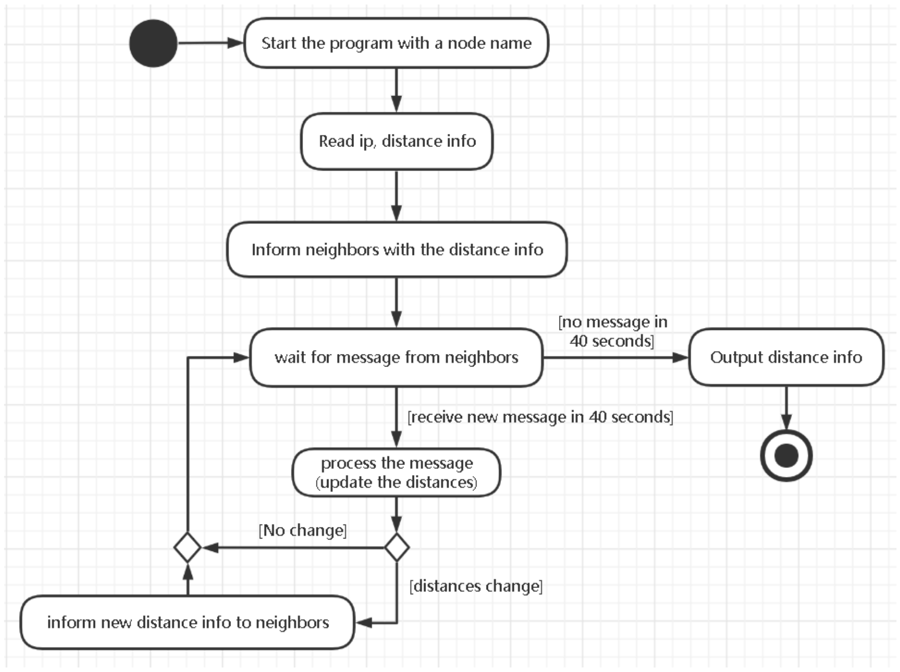
The fourth step:

If the distance_update variable is set to 1, the node needs to notify every neighbor of the DV, then distance_update should be set to 0. If the distance_update is 0 and the received operation code is 0, just send the DV to the neighbor who send this message. In other cases, there is no need to send the DV.

The fifth step:

In the implementation below, the intervals between getting new messages and notifying peers should be relatively short since the number of node is small. If there are still some updates, the node will receive message in 40 seconds. No message for 40 seconds means there is no update any more, the program can just terminate.

3.2. Program flow charts


Start the program with a node name
Read ip, distance info
Inform neighbors with the distance info
wait for message from neighbors
[no message in 40 seconds]
Output distance info
[receive new message in 40 seconds]
process the message (update the distances)
[No change]
[distances change]
inform new distance info to neighbors

3.3. Programming skills

The programming skill I use in this project is exception handling. I use settimeout method to set a time limit for the socket. For my project, the timeout is 40 seconds. Without it, after calling recvfrom method, the project will be blocked until it receives a message. Using it, after 40 seconds, it will throw an exception (timeout), then I can use try-except-else to output the result and terminate when exception happens.

3.4. Difficulties

1. In the beginning, after finishing reading files, I just write the information info DV and send to neighbors without returning DV of neighbors. But there exists a situation that if the DV of the finally started neighbor does not cause any change in the DV of its neighbors, it will get no message to update its DV.
   Solution: At first, I use a very brute force method. That is to send the DV to its neighbors frequently. Then every DV can have chance to be sent to neighbors. Therefore, this problem is solved.
   But there raises another problem. Using brute force method will cause many redundancies since this project does not need to check whether the neighbor is offline or not. The final

method is to use the operation code to mark the greeting message. When neighbors receive this kind of message, no matter there is any change to their DVs or not, they should send their DVs to this node.

4. Testing

4.1. Testing plan

The testing environment is one virtual machine with Tiny Core Linux. Its memory size is 512 MB. The node number will be less than 10. I made three tests for this project. Two with 4 nodes (one normal case and one special case) and one with 10 nodes.
A local port number is assigned to each node. The neighbors' IP info and the IP info for itself is stored in node_name+'_ip.json'. The direct distances to neighbors are stored in node_name+'_distance.json'. All the json files are in the current working directory.

4.2. Testing results

1. The first test:

   The first test is based on Figure 1. The test result for four nodes (In order of u,v,w,x, from up to bottom) are in Figure 2.

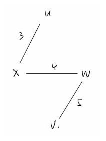
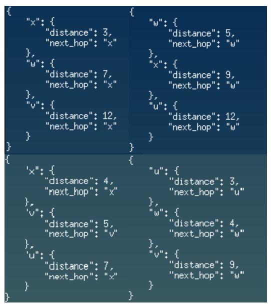


Figure 1: test 1 graphic



```
{                              {
    "x": {                         "w": {
        "distance": 3,                 "distance": 5,
        "next_hop": "x"                "next_hop": "w"
    },                             },
    "w": {                         "x": {
        "distance": 7,                 "distance": 9,
        "next_hop": "x"                "next_hop": "w"
    },                             },
    "v": {                         "u": {
        "distance": 12,                "distance": 12,
        "next_hop": "x"                "next_hop": "w"
    }                              }
}                              }
{                              {
    'x": {                         "u": {
        "distance": 4,                 "distance": 3,
        "next_hop": "x"                "next_hop": "u"
    },                             },
    'v": {                         "w": {
        "distance": 5,                 "distance": 4,
        "next_hop": "v"                "next_hop": "w"
    },                             },
    'u": {                         "v": {
        "distance": 7,                 "distance": 9,
        "next_hop": "x"                "next_hop": "w"
    }                              }
}                              }
```

Figure 2: test 1 result

2. The second test:

   The second test is based on Figure 3. The test result for four nodes (In order of u,v,w,x, from up to bottom) are in Figure 4.

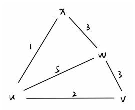
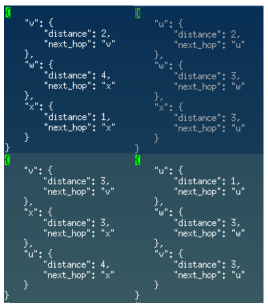


Figure 3: test 2 graphic



```
{
    "v": {
        "distance": 2,
        "next_hop": "v"
    },
    "w": {
        "distance": 4,
        "next_hop": "x"
    },
    "x": {
        "distance": 1,
        "next_hop": "x"
    }
}
```

```
{
    "u": {
        "distance": 2,
        "next_hop": "u"
    },
    "w": {
        "distance": 3,
        "next_hop": "w"
    },
    "x": {
        "distance": 3,
        "next_hop": "u"
    }
}
```

```
{
    "v": {
        "distance": 3,
        "next_hop": "v"
    },
    "x": {
        "distance": 3,
        "next_hop": "x"
    },
    "u": {
        "distance": 4,
        "next_hop": "x"
    }
}
```

```
{
    "u": {
        "distance": 1,
        "next_hop": "u"
    },
    "w": {
        "distance": 3,
        "next_hop": "w"
    },
    "v": {
        "distance": 3,
        "next_hop": "u"
    }
}
```

Figure 4: test 2 result

3. The third test:

   The third test is a 12-node test case found on the internet. I write another python program to compare my answer with the reference answer (compare two dictionaries).

Using these cases, the implementation can be proved to be true.

5. Conclusion

In conclusion, the aim of this project is to implement Bellman-Ford distance vector algorithm for router using Python Socket networking programming which can give the right paths. The project can be an efficient way to understand the basic knowledge behind the Bellman-Ford distance vector algorithm. For this project, there are many improvements that I can make in the future: 1. Try to take neighbor offline into consideration. In this situation, the node needs to send its DV from time to time and need to get answer from neighbors, which means the protocol need to be improved. 2. For real network environment, the router should always be on under normal condition. I should simulate this in my project. 3. The direct distances to neighbors would also be changed.

To make this project more generic to solve many problems in real life, there is still a long way to go.

Reference

[1]M. Corson and A. Ephremides, "A distributed routing algorithm for mobile wireless networks", Wireless Networks, vol. 1, no. 1, pp. 61-81, 1995. Available: 10.1007/bf01196259.
[2]S. Kulkarni and G. Rao, "Mobility and Energy-Based Performance Analysis of Temporally Ordered Routing Algorithm for Ad Hoc Wireless Network", IETE Technical Review, vol. 25, no. 4, p. 222, 2008. Available: 10.4103/0256-4602.42815.