Elaborato di Reti di Telecomunicazioni

Daniel Ariyo

December 2024

1 Introduzione

Come progetto di Reti di Telcomunicazioni ho deciso di scegliere la traccia riguardante la simulazione di un Protocollo di Routing tramite Python. L'obiettivo quindi era dato un grafo pesato in cui i nodi rappresentavano i router e gli archi il costo delle connessioni fra loro, implementare il Distance Vector Routing, per cui ogni nodo data una tabella di routing iniziale tramite varie iterazioni l'aggiornasse fino ad ottenere una tabella ottimale.

2 Classe Node

La classe Node definisce un generico nodo che appartiene al grafo. La classe ha al suo interno 3 variabili:

- name, che definisce il nome del nodo
- routing_table, variabile di tipo Dictionary che mi definisce la vera e propria tabella di routing del nodo
- neighbour, variabile anch'essa di tipo Dictionary che definisce i nodi vicini del mio nodo

Una variabile di tipo Dictionary possiede al suo interno coppie chiave - valore, per routing_table si avrè per ogni chiave che identifica il nodo ci sarà un valore che identifica invece il costo.
Il metodo di default _init_ inizializza un nodo e le sue variabili appena descritte., mentre _str_ fornisce un output a video del nome del nodo e della sua routing table.
Il metodo add_neighbor consente di aggiungere un nuovo vicino al nodo. Vengono aggiornati sia il dizionario neighbors (che memorizza il nome del vicino e il costo per raggiungerlo) che la routing_table, aggiungendo una rotta diretta verso il vicino con il costo specificato.
Il metodo update_routing_table è il cuore del protocollo di routing. Esso permette a un nodo di aggiornare la propria tabella di routing in base alle informazioni ricevute da un vicino: Il nodo esamina tutte le destinazioni nella tabella di routing del vicino. Se la destinazione non è il nodo stesso, calcola il new_cost, che

```python
from collections import defaultdict
class Node:
    def __init__(self, name):
        self.name = name
        self.routing_table = {name: 0}  # Initialize distance to itself as 0
        self.neighbors = {}  # Directly connected neighbors {neighbor: cost}

    def add_neighbor(self, neighbor, cost):
        self.neighbors[neighbor.name] = cost
        self.routing_table[neighbor.name] = cost

    def update_routing_table(self, neighbor):
        updated = False
        for dest, neighbor_cost in neighbor.routing_table.items():
            if dest == self.name:
                continue
            new_cost = self.neighbors[neighbor.name] + neighbor_cost
            if dest not in self.routing_table or new_cost < self.routing_table[dest]:
                self.routing_table[dest] = new_cost
                updated = True
        return updated

    def __str__(self):
        return f"Node {self.name}, Routing Table: {self.routing_table}"
```

Figure 1: Enter Caption

è la somma del costo per raggiungere il vicino e il costo per raggiungere la destinazione tramite il vicino. Se la destinazione non è ancora presente nella tabella del nodo, o se il nuovo costo è inferiore al costo attuale per quella destinazione, la tabella viene aggiornata. Il metodo restituisce True se la tabella è stata aggiornata, altrimenti False.

3 Funzione simulate_distance_vector

La funzione simulate_distance_vector simula il comportamento del protocollo di routing Distance Vector. Essa prende in input una lista di nodi e aggiorna iterativamente le loro tabelle di routing fino a quando tutte le tabelle convergono . Ogni nodo scambia informazioni con i suoi vicini. Per ogni vicino, il nodo aggiorna la sua tabella di routing. Se un nodo ha aggiornato la propria tabella (cioè, se update_routing_table restituisce True), la variabile converged viene impostata su False, indicando che la rete non è ancora stabile e la simulazione deve continuare.

```python
def simulate_distance_vector(nodes):
    converged = False
    iteration = 1
    while not converged:
        print(f"Iteration {iteration}:")
        converged = True
        for node in nodes:
            for neighbor_name in node.neighbors:
                neighbor = next(n for n in nodes if n.name == neighbor_name)
                if node.update_routing_table(neighbor):
                    converged = False
        for node in nodes:
            print(node)
        print("-" * 40)
        iteration += 1
```

Figure 2: Enter Caption

4 Conclusione

Dopo aver definito la classe Node e la funzione simulate_distance_vector vengono creati i nodi e la rete topologica per poi essere dati in input alla funzione descritta sopra, dando come riusltato i distance vector di tutti i nodi presenti all'interno della rete.