Simulazione di Protocollo di Routing

Cristian Morbidelli

December 11, 2024

Contents

1 Introduzione

Il progetto implementa una rete simulata in cui ogni nodo calcola le rotte più brevi verso tutti gli altri nodi usando le informazioni ricevute dai propri vicini. Questo approccio è basato sul protocollo **Distance Vector Routing**, ampiamente utilizzato nei sistemi di rete reali.

Gli obiettivi principali del progetto sono:

- Implementare nodi con tabelle di routing inizializzate localmente.
- Gestire dinamicamente i collegamenti tra i nodi con costi specifici.
- Simulare la convergenza delle tabelle di routing verso rotte ottimali.
- Fornire una rappresentazione chiara dei risultati ottenuti.

2 Descrizione del Codice

Il codice si compone di due classi principali:

- **Node**: rappresenta un nodo nella rete, con una tabella di routing locale e una lista di vicini.
- **Network**: rappresenta l'intera rete, gestendo i nodi, i collegamenti e la simulazione del protocollo.

2.1 Classe Node

La classe **Node** include i seguenti attributi e metodi principali:

- **add_neighbor(neighbor, cost)**: aggiunge un vicino con un costo specificato.

```
def add_neighbor(self, neighbor, cost):
    self.neighbors[neighbor] = cost
    self.routing_table[neighbor] = {"cost":
        cost, "next_hop": neighbor}
```

Listing 1: Aggiunta di un vicino al nodo

- **update_routing_table(neighbor_table, neighbor_name)**: aggiorna la tabella di routing in base alle informazioni ricevute da un vicino.

```
def update_routing_table(self,
   neighbor_table, neighbor_name):
    updated = False
    for dest, data in neighbor_table.items
       ():
        if dest == self.name:  # Ignorare
           il costo verso se stessi.
            continue
```

```
6           new_cost = self.neighbors[
               neighbor_name] + data["cost"]
7           if dest not in self.routing_table
               or new_cost < self.routing_table
               [dest]["cost"]:
8             self.routing_table[dest] = {"
                 cost": new_cost, "next_hop":
                  neighbor_name}
9             updated = True
10      return updated
```

Listing 2: Aggiornamento della tabella di routing

- **print_routing_table()**: stampa la tabella di routing del nodo.

2.2 Classe Network

La classe **Network** include:

- **add_link(node1, node2, cost)**: crea un collegamento bidirezionale tra due nodi con un determinato costo.

```
1 def add_link(self, node1, node2, cost):
2     self.nodes[node1].add_neighbor(node2,
         cost)
3     self.nodes[node2].add_neighbor(node1,
         cost)
```

Listing 3: Creazione di un collegamento bidirezionale

- **simulate_routing()**: implementa l'algoritmo di convergenza per calcolare le rotte più brevi.

```
1 def simulate_routing(self):
2     converged = False
3     while not converged:
4         converged = True
5         for node_name, node in self.nodes.
             items():
6           for neighbor_name in node.
               neighbors:
7             neighbor = self.nodes[
                 neighbor_name]
8             if node.
                 update_routing_table(
                 neighbor.routing_table,
                 neighbor_name):
```

```
converged = False
```

Listing 4: Simulazione del processo di convergenza

3 Esecuzione e Output

Lo script è progettato per simulare il routing su una rete con i seguenti nodi e collegamenti iniziali:

- **Nodi**: `A`, `B`, `C`, `D`
- **Collegamenti iniziali**:
  - `A - B`: costo 1
  - `A - C`: costo 4
  - `B - C`: costo 2
  - `C - D`: costo 1

L'output iniziale delle tabelle di routing è mostrato dopo l'aggiunta dei collegamenti. La simulazione procede calcolando iterativamente i percorsi più brevi fino alla convergenza. Ogni nodo aggiorna la propria tabella basandosi sulle informazioni ricevute dai vicini.

4 Conclusioni

Il progetto dimostra come implementare e simulare il protocollo **Distance Vector Routing** utilizzando Python. La rete convergerà verso uno stato ottimale in cui tutti i nodi conoscono i percorsi più brevi verso ogni altro nodo. Lo script può essere ulteriormente esteso per gestire dinamicamente cambiamenti nei collegamenti e simulare fallimenti di nodi o percorsi.