Progetto Laboratorio Reti di Telecomunicazioni A.A. 2024-25

Nicholas Ricci - Mat: 0001071525

December 2024

Introduzione

L'obiettivo è sviluppare uno script Python che simuli un protocollo di routing di base, come il **Distance Vector Routing**. Quindi si deve implementare il meccanismo per aggiornare le tabelle di routing tra i nodi della rete, calcolando al contempo i percorsi più brevi.

Classe: Nodo

Il codice definisce una classe **Node**, che rappresenta un nodo all'interno di una rete. Ogni nodo ha un identificativo univoco (*node_id*) e una lista di vicini (*neighbors*), rappresentati da un dizionario in cui le chiavi sono gli ID dei nodi vicini e i valori sono le distanze a questi nodi.

```python
# Node class represents each node in the network with its unique id, neighbors, and routing table
class Node:
    def __init__(self, node_id, neighbors):
        """
        Initialize a Node object.

        Parameters:
        node_id (int): The unique identifier for the node.
        neighbors (dict): A dictionary where keys are neighbor node IDs and values are the distances to those neighbors.
        """
        self.node_id = node_id  # Set the node's unique ID
        self.neighbors = neighbors  # Set the node's neighbors and their respective distances
        self.routing_table = {node_id: (0, None)}  # Initialize the routing table with the node's ID, distance 0, and no next hop

        # Add entries to the routing table for each neighbor with their distance and next hop (which is the neighbor itself)
        for neighbor, distance in neighbors.items():
            self.routing_table[neighbor] = (distance, neighbor)
```

Il costruttore della classe Node crea un nodo con un identificatore univoco,

una lista di vicini con le rispettive distanze, e una tabella di routing iniziale che include il nodo stesso e i suoi vicini diretti.

La tabella di routing è un dizionario che mappa ogni destinazione (sia essa un nodo vicino o il nodo stesso) alla distanza e al prossimo nodo da attraversare.

Metodo: update_routing_table

La classe **Node** implemente il metodo *update_routing_table* che ha come scopo quello di aggiornare la tabella di routing del nodo corrente confrontando le rotte esistenti con quelle contenute in una tabella di routing ricevuta. L'aggiornamento avviene se viene trovata una rotta più breve per raggiungere una determinata destinazione.

```python
def update_routing_table(self, table, source_id):
    """
    Update the node's routing table based on the received routing table from another node.

    Parameters:
    table (dict): The routing table received from another node.
    source_id (str): The ID of the node that sent the routing table.
    """
    # Get the distance to the source node (if it's a neighbor) or set it to infinity
    neighbor_distance = self.neighbors.get(source_id, float('inf'))

    # Update the current node's routing table based on the received information
    for destination, (distance, next_hop) in table.items():
        if destination != self.node_id:

            if next_hop == self.node_id:
                continue

            # Calculate the new distance by adding the neighbor's distance
            new_distance = distance + neighbor_distance

            # If the new distance is shorter, update the routing table
            if new_distance < self.routing_table.get(destination, (float("inf"), None))[0]
            :
                self.routing_table[destination] = (new_distance, source_id)
```

Analisi del codice: Il codice scorre la tabella ricevuta per identificare l'ID del nodo che ha inviato la tabella (*other_node_id*). Questo è dedotto dal fatto che, nella tabella ricevuta, una destinazione avrà *next_hop* impostato a *None* (il nodo stesso).

Viene determinata la distanza dal nodo corrente al nodo vicino utilizzando il dizionario dei vicini del nodo corrente. Se il nodo vicino non è presente tra i vicini diretti, la distanza viene impostata a infinito (*float('inf')*).

Si scorre nuovamente la tabella ricevuta e per ogni destinazione diversa dal nodo corrente si calcola una nuova distanza (*new_distance*) come somma della distanza dal nodo corrente al nodo vicino (*neighbor_distance*) e la distanza del vicino alla destinazione (*distance*).

Si confronta la nuova distanza con la distanza attualmente registrata nella

tabella di routing del nodo, se tale destinazione non è presente nella tabella la distanza è uguale ad infinito in quanto nuova destinazione.

Se la nuova distanza è più breve, la tabella di routing viene aggiornata con la nuova distanza e con l'ID del nodo vicino (*other_node_id*) come prossimo salto.

Conclusione: Il metodo implementa quindi la logica principale del protocollo **Distance Vector Routing**: propagare informazioni sui percorsi nella rete e aggiornare le tabelle locali con i percorsi più efficienti. Grazie a questa logica, ogni nodo può costruire una vista aggiornata delle distanze minime verso tutti gli altri nodi.

Metodo: get_routing_table

Il metodo *get_routing_table* è un'implementazione sicura all'interno della classe **Node** che permette di accedere alla tabella di routing senza compromettere i dati originali del nodo. Questo approccio garantisce incapsulamento e protezione dei dati.

```
def get_routing_table(self):
    """
    Get a deep copy of the node's routing table.

    Returns:
    dict: A copy of the node's routing table.
    """
    return copy.deepcopy(self.routing_table)  # Return a deep copy of the routing table
```

Main Loop

La funzione *simulate_routing* si occupa di simulare il funzionamento del protocollo **Distance Vector Routing**. Il "main loop" infatti permette ai nodi di aggiornare progressivamente le proprie tabelle di routing, condividendo informazioni con i vicini e scoprendo nuove rotte fino a ottenere una visione completa della rete.

```python
# Simulate the routing process
def simulate_routing(nodes):
    """
    Simulate the Distance Vector Routing process for all nodes in the network.

    Parameters:
    nodes (list): A list of Node objects that represents the network.
    """
    # Run the simulation for a fixed number of iterations (convergence)
    for i in range(len(nodes)):
        print(f"--- Iteration {i + 1} ---")

        # Update routing tables for all nodes
        for current_node in nodes:
            for other_node in nodes:
                if current_node.node_id != other_node.node_id:
                    # Update the current node's routing table with the routing table from
                    another node
                    current_node.update_routing_table(other_node.get_routing_table())

        # Print the routing table of each node after the update
        for node in nodes:
            node.print_routing_table()
```

La simulazione si ripete per un numero di iterazioni pari al numero di nodi nella rete. Questo numero è sufficiente affinchè le informazioni sui percorsi si propaghino completamente (convergenza).

Durante ciascuna iterazione, ogni nodo invia la propria tabella di routing (vedi: *"Metodo: get_routing_table"*) agli altri nodi. Ogni nodo riceve queste tabelle, le analizza e calcola eventuali nuove distanze (vedi: *"Metodo: update_routing_table"*).

Alla fine di ogni iterazione, la tabella di routing di ciascun nodo viene stampata permettendo di osservare l'evoluzione dei dati e verificare come i percorsi migliori vengono scoperti passo dopo passo.