Simulazione di Protocollo di Routing

Bartolini Riccardo
riccardobartolini@studio.unibo.it
Matricola: 0001068901

Fabbri Gianmarco
gianmarcofabbri3@studio.unibo.it
Matricola: 0001069003

1 Introduzione

Il progetto simula un semplice protocollo di routing basato sull'algoritmo **Distance Vector Routing**. L'obiettivo è implementare la logica di aggiornamento delle tabelle di routing tra nodi in una rete, calcolando le rotte più brevi tra di essi. La simulazione converge quando tutte le tabelle di routing diventano stabili.

2 Struttura del Codice

Il progetto è suddiviso in tre moduli principali:

1. `node.py`

Contiene la classe `Node`, che rappresenta un nodo nella rete. Ogni nodo ha:

- Una lista di vicini diretti con il costo associato.
- Una tabella di routing che memorizza:
  - La destinazione.
  - Il costo per raggiungere la destinazione.
  - Il prossimo hop verso la destinazione.

Metodi principali:

`add_neighbor(neighbor, cost)` Aggiunge un vicino diretto al nodo e aggiorna la tabella di routing.

```
def add_neighbor(self, neighbor: 'Node', cost: int):
    if neighbor in self.neighbors:
        if self.neighbors[neighbor] != cost:
            raise ValueError(f"Il vicino {neighbor.name}    gi
               definito con un costo {self.neighbors[neighbor]}."
               )
    self.neighbors[neighbor] = cost
    self.routing_table[neighbor.name] = (cost, neighbor.name)
```

`remove_neighbor(neighbor)` Simula la perdita di una connessione, aggiornando la tabella di routing.

```
def remove_neighbor(self, neighbor: 'Node'):
    if neighbor in self.neighbors:
        del self.neighbors[neighbor]
        self.routing_table.pop(neighbor.name, None)
```

`update_routing_table()` Aggiorna la tabella di routing basandosi sulle informazioni ricevute dai vicini. Restituisce `True` se la tabella è stata modificata, `False` altrimenti.

```
def update_routing_table(self) -> bool:
    if not self.neighbors:
        print(f"Attenzione: il nodo {self.name} non ha vicini.")
        return False
    updated = False
    for neighbor, cost_to_neighbor in self.neighbors.items():
        neighbor_table = neighbor.routing_table
        for dest, (cost_to_dest, _) in neighbor_table.items():
            if dest == self.name:
                continue
            new_cost = cost_to_neighbor + cost_to_dest
            current_cost = self.routing_table.get(dest, (float('
               inf'), None))[0]
            if new_cost < current_cost:
                self.routing_table[dest] = (new_cost, neighbor.
                   name)
                updated = True
    return updated
```

`print_routing_table_to_string()` Restituisce la tabella di routing come stringa leggibile.

```
def print_routing_table_to_string(self) -> str:
    output = [f"Tabella di routing per il nodo {self.name}:"]
    output.append(f"{'Destinazione':<15}{'Costo':<10}{'Prossimo
       Hop':<15}")
    for dest, (cost, next_hop) in sorted(self.routing_table.items
       ()):
        output.append(f"{dest:<15}{cost:<10}{next_hop:<15}")
    output.append("-" * 40)
    return "\n".join(output)
```

2. network.py

Contiene un'unica funzione:

`simulate_network(nodes, output_file=None)` Gestisce il processo iterativo in cui ogni nodo aggiorna la propria tabella di routing basandosi sulle informazioni ricevute dai vicini. Durante ogni iterazione, le tabelle di routing aggiornate vengono stampate per monitorare l'evoluzione della rete. La simulazione termina quando tutte le tabelle diventano stabili.

```
def simulate_network(nodes: List[Node], output_file: str = None):
    iteration = 0
    output = []
    while True:
        output.append(f"\n--- Iterazione {iteration} ---")
        updated = False
        for node in nodes:
            if node.update_routing_table():
                updated = True
        for node in nodes:
            output.append(node.print_routing_table_to_string())
        if not updated:
            output.append("Le tabelle di routing sono stabili.")
            break
        iteration += 1
    if output_file:
        with open(output_file, 'w') as f:
            f.writelines("\n".join(output))
    else:
        print("\n".join(output))
```

3. main.py

È il punto di ingresso del programma. Configura i nodi, definisce le connessioni tra di essi e avvia la simulazione.

L'algoritmo Distance Vector Routing si basa sui seguenti principi:

1. **Inizializzazione:** ogni nodo conosce solo i costi per raggiungere i suoi vicini diretti.
2. **Scambio di informazioni:** ad ogni iterazione, i nodi condividono le proprie tabelle di routing con i vicini.
3. **Aggiornamento delle tabelle:** per ogni destinazione ricevuta dai vicini, il nodo valuta se:
   - Esiste un percorso più economico.
   - È necessario aggiornare la tabella di routing.
4. **Convergenza:** la simulazione si ferma quando le tabelle di routing diventano stabili, ovvero non si verificano ulteriori aggiornamenti.

3 Simulazione

Nell'esempio fornito, la rete è costituita da quattro nodi: A, B, C e D. Le connessioni dirette tra i nodi sono definite con i seguenti costi:

- A $\rightarrow$ B (1)
- A $\rightarrow$ C (5)
- B $\rightarrow$ C (2)
- B $\rightarrow$ D (4)
- C $\rightarrow$ D (1)

Il programma esegue iterazioni fino alla convergenza: Alla prima iterazione (**Iterazione 0**), ogni nodo conosce solo i vicini diretti. Nelle iterazioni successive: i nodi iniziano a scoprire percorsi indiretti verso altre destinazioni, aggiornando gradualmente le loro tabelle di routing. Quando le tabelle non cambiano più, la simulazione si ferma.

Output

L'output della simulazione è costituito dalle tabelle di routing aggiornate a ogni iterazione, salvate in un file di testo. Di seguito un esempio dimostrativo di una tabella di routing generata durante la simulazione.

Destinazione	**Costo**	**Prossimo Hop**
A	0	A
B	1	B
C	3	B
D	5	B

Table 1: Tabella di routing per il nodo A.

4 Cenni teorici

Il protocollo **Distance Vector Routing** è un algoritmo distribuito che consente ai nodi di una rete di calcolare i percorsi ottimali verso tutte le altre destinazioni.

I principi fondamentali alla base di questo algoritmo includono:

1. **Propagazione locale delle informazioni:** ogni nodo condivide la propria tabella di routing con i vicini diretti.
2. **Aggiornamento iterativo:** le tabelle vengono aggiornate iterativamente, utilizzando l'equazione di Bellman-Ford:
$$d(v) = \min_{u \in neighbors} \{d(u) + cost(u, v)\}$$
dove $d(v)$ è il costo minimo per raggiungere il nodo v.
3. **Convergenza:** la rete raggiunge la stabilità quando nessuna tabella viene più modificata.

Questo protocollo ha alcune limitazioni ben documentate, tra cui il problema dei "conti alla rovescia" (*count to infinity*), che si verifica in presenza di cambiamenti drastici nella rete.

Possibili miglioramenti

Nonostante la semplicità ed efficacia del Distance Vector Routing, esistono diverse aree in cui il protocollo può essere migliorato:

- **Risoluzione del problema dei conti alla rovescia:** Si possono implementare tecniche come:
  - *Split horizon*: impedisce l'invio di informazioni di routing al nodo da cui sono state ricevute.
  - *Route poisoning*: assegna un costo infinito a rotte non più valide, accelerando la convergenza.
- **Rilevamento rapido di cambiamenti:** L'adozione di notifiche di aggiornamento immediate (*triggered updates*) consente alla rete di adattarsi più velocemente ai cambiamenti nei collegamenti.
- **Riduzione del traffico di controllo:** Utilizzare algoritmi che riducono la frequenza degli aggiornamenti periodici senza compromettere la reattività della rete.
- **Scalabilità:** Per reti di grandi dimensioni, l'adozione di protocolli alternativi, come il **Link State Routing**, può migliorare le prestazioni, garantendo tempi di convergenza più rapidi e una maggiore efficienza.
- **Gestione dinamica dei costi:** L'integrazione di metriche avanzate (ad esempio, latenza, larghezza di banda o congestione) consentirebbe al protocollo di scegliere percorsi più affidabili e performanti.

Questi miglioramenti non solo rendono il protocollo più robusto, ma ne aumentano anche la scalabilità e l'adattabilità in ambienti complessi e dinamici.

5 Conclusioni

La simulazione ha evidenziato come il protocollo Distance Vector Routing consenta ai nodi di calcolare percorsi ottimali in maniera distribuita, basandosi esclusivamente su informazioni locali scambiate con i vicini diretti. Ogni nodo aggiorna iterativamente la propria tabella di routing, costruendo progressivamente una rappresentazione completa dei costi minimi e dei percorsi ottimali verso ogni altra destinazione nella rete.

Di seguito sono riportati i principali aspetti emersi dallo studio del protocollo:

- **Semplicità dell'approccio:**
  Il Distance Vector Routing si distingue per la sua implementazione intuitiva, che sfrutta un meccanismo iterativo di scambio di informazioni tra nodi vicini. Questo approccio distribuito lo rende adatto a reti decentralizzate, dove ogni nodo può calcolare autonomamente le proprie rotte.

- **Progresso graduale verso la convergenza:**
  Il comportamento iterativo del protocollo garantisce che le tabelle di routing si stabilizzino nel tempo. Durante ogni iterazione, i nodi aggiornano i percorsi utilizzando le informazioni più recenti ricevute dai vicini, avvicinandosi progressivamente a una configurazione stabile.

- **Adattabilità ai cambiamenti:**
  La rete può rispondere a modifiche dinamiche, come l'aggiunta o la rimozione di collegamenti e variazioni nei costi. Tuttavia, l'adattamento richiede più iterazioni e può temporaneamente introdurre percorsi sub-ottimali. Questo evidenzia la flessibilità del protocollo, ma anche i suoi limiti in termini di rapidità di convergenza.

- **Sfide di scalabilità:**
  Sebbene il protocollo sia efficace in reti di dimensioni moderate, un aumento del numero di nodi o una maggiore frequenza di cambiamenti nella rete possono comportare un incremento significativo del tempo necessario per la convergenza. Per applicazioni più complesse, è opportuno valutare protocolli alternativi o miglioramenti specifici al Distance Vector Routing.

La simulazione dimostra quindi la validità del Distance Vector Routing per reti relativamente statiche e di dimensioni contenute, sottolineando al contempo i limiti del protocollo in ambienti più dinamici o complessi.