Progetto python

Questo script simula un sistema di routing chiamato **RIP** basato sull'algoritmo di **Distance-Vector** , un approccio distribuito in cui i nodi (router) costruiscono le loro tabelle di instradamento basandosi sulle informazioni condivise dai nodi vicini.

Il file di input definisce una rete in formato testuale, con ogni nodo e i suoi vicini diretti. Lo script costruisce il grafo della rete, simula la propagazione delle informazioni di routing e calcola le tabelle di instradamento finali per ogni nodo.

Algoritmo Distance Vector

Inizialmente la tabella di routing di un nodo ha solamente la rotta per se stesso e per i propri vicini, per i quali può conoscere la distanza in maniera immediata.

Periodicamente ogni nodo comunica la propria tabella di routing a tutti i suoi vicini i quali aggiornano la propria confrontando le 2 tabelle. Se un nodo A riceve una rotta dal vicino B per una destinazione lontana D, viene sommata la distanza tra il A e B, mentre il numero di hops necessari aumenta di uno rispetto a quello ricevuto da B. Se con questi dati la rotta sarebbe conveniente rispetto a quella attuale di A, allora viene sostituita e il prossimo hop per raggiungere D diventerà B. Si può riassumere l'algoritmo con il seguente pseudocodice:

if (distanza_BD + distanza_AB < distannza_AD) then

    distanza_AD = distanza_BD + distanza_AB

    next_hopAD = B

Soluzioni e Ottimizzazioni

1. Split Horizon

Lo **Split Horizon** è una tecnica per prevenire i cicli di routing. Nel contesto di questo script, si realizza eliminando dalla tabella di routing condivisa le rotte che utilizzano il vicino come hop successivo. Questo evita che un nodo comunichi a un vicino una rotta che il vicino stesso gli ha originariamente comunicato, riducendo significativamente il rischio di cicli infiniti.

Implementazione nello script:

```
[(d, r) for d, r in self.routing_table.items() if r.next != neighbor.name]
```

Questo filtro elimina le rotte per cui il next hop coincide con il vicino a cui si sta inviando la tabella.

2. Limite Massimo di Salti

Il limite massimo di salti è un meccanismo di sicurezza per evitare che le rotte vengano propagate indefinitamente, che vengano accettate rotte troppo dispendiose e a ridurre il tempo di convergenza. Se una rotta comporta un numero di salti maggiore del limite questa viene scartata. Il limite è definito nella costante `MAX_HOP` impostata a 15.

3. Scoperta di nodi sconosciuti

Quando un nodo riceve una rotta per una destinazione non presente nella propria routing table deve necessariamente aggiungerla. Alcune implementazioni del distance-vector suggeriscono di porre una distanza iniziale pari a infinito per ogni nodo destinazione esistente nel grafo, così qualsiasi rotta scoperta avrà distanza minore e sarà sostituita.

Diversamente nel mio esistono solo le rotte per i nodi raggiungibili, ogni nodo scoperto viene aggiunto in coda alla tabella, ne consegue che la routing table non è ordinata, ma credo che sia più realistico che inizialmente il nodo non conosce quanti e quali nodi siano presenti nella rete.

Struttura del Codice

1. Classi Principali

Route

Definisce una rotta da associare a una destinazione con i seguenti attributi:

- `distance`**:** La distanza cumulativa per raggiungere la destinazione.
- `hops`**:** Il numero di hop necessari.
- `next`**:** Il nodo successivo lungo il percorso.

Node

Rappresenta un nodo (router) nella rete.

Attributi:

- `name`: Nome del nodo.
- `neighbors`: Lista dei nodi vicini.
- `routing_table`: Mappa da nome destinazione a rotta

Metodi Principali:

- `addNeighbor(neighbor, distance)`: Aggiunge un vicino al nodo e aggiorna la tabella di routing con una rotta diretta.
- `updateRoutingTable(neighbor, neighbor_routing_table)`: Aggiorna la tabella di routing con le informazioni ricevute da un vicino.

- `updateNeighbors()`: Propaga la tabella di routing del nodo a tutti i vicini

2. Funzione di Inizializzazione

`networkInit()`: Legge il file di input e costruisce il relativo grafo della rete. Ogni nodo riceve i riferimenti ai propri vicini e le rispettive distanze.

Formato input: <nodo1>;<vicino1>:<distanza1>,<vicino2>:<distanza2>: …
<nodo2>; …

3. Simulazione

La simulazione della rete avviene iterativamente:

1. Ogni nodo aggiorna i vicini propagando la propria tabella di routing.
2. Le iterazioni continuano su tutti i nodi finché per un intero ciclo non avviene neanche una modifica alle rotte.

In questo caso la simulazione può essere interrotta perché non sono previste modifiche alla rete in fase di esecuzione, come può invece avvenire nel vero mondo del routing IP.

4. Output

Il programma stampa:

- **Cicli per la convergenza:** Numero di iterazioni necessarie per ottenere tabelle di routing stabili.
- **Tabelle di Routing Finali:** La tabella di routing completa per ogni nodo.

Esempio di output

```
Cicli per la convergenza: 5
Routing table per nodo A:
  A -> A ; distanza: 0, salti: 0
  B -> B ; distanza: 5, salti: 1
  C -> C ; distanza: 2, salti: 1
  ...
Routing table per nodo B:
  …
```

Conclusioni

Questo semplice algoritmo funziona bene per reti piccole e molto stabili, dove i collegamenti sono interrotti raramente, ma presenta delle problematiche irrisolvibili in maniera efficiente se in particolari grafi i collegamenti possono scomparire in qualsiasi momento. Infatti il protocollo RIP non è attualmente utilizzato per l'instradamento IP in rete perché si basa sul distance-vector che non risolve queste problematiche esistenti nella rete reale.