documentation

Script Documentation - Distance Vector Routing

- Autore: Mosconi Simone
- Matricola: 0001081246
- Anno accademico: 2024/25
- Mail istituzionale: simone.mosconi2@studio.unibo.it

Repository: https://github.com/SimoneM-17/DVR-Protocol-Simulator

---

Introduzione

Il sistema implementa una simulazione del protocollo **Distance Vector Routing**, uno degli algoritmi fondamentali utilizzati nei protocolli di routing dinamico come RIP (Routing Information Protocol).

Obiettivi del Sistema

- Dimostrare il funzionamento dell'algoritmo Distance Vector
- Visualizzare il processo di convergenza delle tabelle di routing
- Permettere l'analisi di diverse topologie di rete

Istruzioni sul Funzionamento del Programma

Avvio e Configurazione

1. **Esecuzione**: `python main.py`
2. **Selezione modalità**: Verrà chiesto di scegliere tra topologia predefinita o personalizzata
3. **Configurazione**: Nel secondo caso occorre specificare manualmente la struttura della rete

Inizializzazione

1. **Creazione tabelle**: Inizializzazione delle tabelle di routing
2. **Visualizzazione iniziale**: Stampa dello stato iniziale
3. **Preparazione simulazione**: Setup delle strutture dati

Simulazione Iterativa

```
Iterazione 1:
  ├── Aggiornamento tabelle
  ├── Verifica cambiamenti
  ├── Visualizzazione stato
  └── Controllo convergenza

Iterazione 2:
  ├── Aggiornamento tabelle
  ├── Verifica cambiamenti
  ├── Visualizzazione stato
  └── Controllo convergenza

...continua fino alla convergenza
```

Completamento

1. **Rilevamento convergenza**: Nessun cambiamento nelle tabelle
2. **Visualizzazione finale**: Stampa delle tabelle finali
3. **Terminazione**: Fine della simulazione

Architettura del Sistema

Il sistema è composto da **tre moduli principali**:

```
Sistema Distance Vector Routing
├── main.py     # Punto di ingresso dell'applicazione
├── network_config.py      # Gestione configurazione di rete
└── routing.py     # Implementazione algoritmo Distance Vector
```

Flusso di Esecuzione

1. **Inizializzazione**: Caricamento del sistema e selezione della modalità
2. **Configurazione**: Definizione della topologia di rete
3. **Simulazione**: Esecuzione dell'algoritmo Distance Vector
4. **Visualizzazione**: Presentazione dei risultati iterativi

---

Moduli del Sistema

1. main.py

Rappresenta il punto di ingresso principale dell'applicazione

Codice Principale:

```
if __name__ == "__main__":
    network_topology = select_network_mode()
    run_distance_vector_simulation(network_topology)
```

2. network_config.py

Scopo: Gestione della configurazione della topologia di rete

Funzioni Principali

`select_network_mode()`

- **Descrizione**: Permette all'utente di scegliere tra topologia predefinita o personalizzata
- **Ritorno**: Dizionario rappresentante la topologia di rete
- **Modalità disponibili**:
  - Topologia predefinita (8 nodi con collegamenti specifici)
  - Topologia personalizzata (configurazione manuale)

`build_custom_topology()`

- **Descrizione**: Acquisisce la configurazione di rete tramite input utente
- **Processo**:
  1. Richiesta numero totale di nodi
  2. Configurazione iterativa di ogni nodo
  3. Definizione dei collegamenti bidirezionali
- **Formato input**: `NodoDestinazione,Costo`

`display_network_structure(topology)`

- **Descrizione**: Visualizza la struttura della rete in formato leggibile
- **Parametri**: `topology` - Dizionario della topologia di rete

Topologia Predefinita

La topologia di esempio include 8 nodi (A-H) con i seguenti collegamenti:

```
A ↔ B (costo: 2)    E ↔ C (costo: 7)
A ↔ C (costo: 5)    E ↔ D (costo: 4)
```

```
B ↔ C (costo: 3)    E ↔ G (costo: 2)
B ↔ D (costo: 1)    F ↔ D (costo: 8)
                    F ↔ G (costo: 6)
                    F ↔ H (costo: 3)
                    G ↔ H (costo: 1)
```

3. routing.py

Scopo: Implementazione dell'algoritmo Distance Vector Routing

Funzioni Principali

`create_initial_tables(topology)`

- **Descrizione**: Inizializza le tabelle di routing per tutti i nodi
- **Logica di inizializzazione**:
  - Distanza verso sé stesso: 0
  - Distanza verso nodi adiacenti: costo del collegamento
  - Distanza verso altri nodi: infinito
- **Ritorno**: Dizionario delle tabelle di routing iniziali

`refresh_distance_tables(topology, distance_tables)`

- **Descrizione**: Aggiorna le tabelle di routing utilizzando l'algoritmo Distance Vector
- **Processo**:
  1. Crea copia profonda delle tabelle attuali
  2. Per ogni nodo, esamina i percorsi attraverso i vicini
  3. Aggiorna le distanze se trova percorsi migliori
- **Ritorno**: Tupla (nuove_tabelle, flag_cambiamento)

`run_distance_vector_simulation(topology, max_cycles=100)`

- **Descrizione**: Esegue la simulazione completa del protocollo
- **Parametri**:
  - `topology` : Topologia di rete
  - `max_cycles` : Limite massimo di iterazioni (default: 100)
- **Processo**:
  1. Inizializzazione delle tabelle
  2. Ciclo iterativo di aggiornamento
  3. Verifica della convergenza
  4. Visualizzazione dei risultati

`display_routing_tables(distance_tables)`

- **Descrizione**: Visualizza le tabelle di routing in formato leggibile
- **Output**: Stampa formattata delle tabelle per ogni nodo

Algoritmo Distance Vector

Principio di Funzionamento

L'algoritmo Distance Vector si basa sul principio di **Bellman-Ford distribuito**:

1. **Inizializzazione**: Ogni nodo conosce solo la distanza verso i suoi vicini diretti
2. **Scambio di informazioni**: I nodi condividono le loro tabelle di routing
3. **Aggiornamento**: Ogni nodo aggiorna le sue distanze basandosi sulle informazioni ricevute
4. **Convergenza**: Il processo continua fino a quando non ci sono più cambiamenti

Formula di Aggiornamento

Per ogni nodo `i`, destinazione `j` e vicino `k`:

```
distanza[i][j] = min(distanza[i][j], distanza[i][k] + distanza[k][j])
```

Condizioni di Convergenza

La convergenza è raggiunta quando:

- Nessuna tabella di routing viene modificata in un'iterazione
- Tutte le distanze rappresentano i percorsi minimi

Strutture Dati

Rappresentazione della Topologia

```
topology = {
    'A': {'B': 2, 'C': 5},      # Nodo A collegato a B (costo 2) e C (costo 5)
    'B': {'A': 2, 'C': 3, 'D': 1},  # Nodo B con collegamenti multipli
    # ... altri nodi
}
```

Tabelle di Routing

```
distance_tables = {
    'A': {'A': 0, 'B': 2, 'C': 5, 'D': float('inf'), ...},
    'B': {'A': 2, 'B': 0, 'C': 3, 'D': 1, ...},
    # ... tabelle per tutti i nodi
}
```

Caratteristiche delle Strutture

- **Grafo non orientato**: Collegamenti bidirezionali
- **Costi positivi**: Tutti i pesi dei collegamenti sono positivi
- **Rappresentazione a dizionario**: Facilita l'accesso e la modifica
- **Gestione dell'infinito**: Utilizzato per distanze non ancora calcolate

---

Esempi di Utilizzo

Esempio 1: Topologia Predefinita

Input:

```
Seleziona la modalità di configurazione della rete:
Scrivi 'predefinita' per la topologia di esempio o 'personalizzata' per
configurarne una: predefinita
```

Output Iniziale:

```
Struttura della rete corrente:
A -> {'B': 2, 'C': 5}
B -> {'A': 2, 'C': 3, 'D': 1}
C -> {'A': 5, 'B': 3, 'E': 7}
...

Creazione delle tabelle di instradamento iniziali:
Tabella del nodo A: {'A': 0, 'B': 2, 'C': 5, 'D': inf, 'E': inf, 'F': inf, 'G':
inf, 'H': inf}
Tabella del nodo B: {'A': 2, 'B': 0, 'C': 3, 'D': 1, 'E': inf, 'F': inf, 'G':
inf, 'H': inf}
...
```

Processo di Convergenza:

```
Ciclo di aggiornamento 1:
Tabella del nodo A: {'A': 0, 'B': 2, 'C': 5, 'D': 3, 'E': 12, 'F': inf, 'G': inf,
'H': inf}
...

Ciclo di aggiornamento 2:
Tabella del nodo A: {'A': 0, 'B': 2, 'C': 5, 'D': 3, 'E': 7, 'F': 11, 'G': 9,
'H': inf}
...

Stato di convergenza raggiunto con successo
```

Esempio 2: Topologia Personalizzata

Input:

```
Seleziona la modalità di configurazione della rete:
Scrivi 'predefinita' per la topologia di esempio o 'personalizzata' per
configurarne una: personalizzata

Configurazione manuale della topologia di rete...
Inserire il numero totale di nodi nella rete: 3
Inserisci l'identificatore del nodo 1: X
Inserisci i collegamenti diretti per il nodo X:
Collegamenti per X (formato: NodoDestinazione,Costo) oppure 'fine' per terminare:
Y,1
Collegamenti per X (formato: NodoDestinazione,Costo) oppure 'fine' per terminare:
fine
Inserisci l'identificatore del nodo 2: Y
Inserisci i collegamenti diretti per il nodo Y:
Collegamenti per Y (formato: NodoDestinazione,Costo) oppure 'fine' per terminare:
Z,2
Collegamenti per Y (formato: NodoDestinazione,Costo) oppure 'fine' per terminare:
fine
Inserisci l'identificatore del nodo 3: Z
Inserisci i collegamenti diretti per il nodo Z:
Collegamenti per Z (formato: NodoDestinazione,Costo) oppure 'fine' per terminare:
fine
```

Risultato:

```
Struttura della rete corrente:
X -> {'Y': 1}
Y -> {'X': 1, 'Z': 2}
Z -> {'Y': 2}

Tabelle finali dopo la convergenza:
Tabella del nodo X: {'X': 0, 'Y': 1, 'Z': 3}
Tabella del nodo Y: {'X': 1, 'Y': 0, 'Z': 2}
Tabella del nodo Z: {'X': 3, 'Y': 2, 'Z': 0}
```

RISOLUZIONE DELLE SFIDE TECNICHE NELL'IMPLEMENTAZIONE DEL SISTEMA

Principali Ostacoli Affrontati Durante lo Sviluppo

Durante la creazione del sistema di simulazione Distance Vector Routing, sono emerse diverse problematiche tecniche.

Identificazione e Controllo dello Stato di Convergenza

La sfida più significativa è stata la progettazione di un sistema affidabile per determinare il momento esatto in cui l'algoritmo raggiunge la stabilità. La soluzione adottata prevede l'utilizzo di un indicatore di stato (`has_changed`) che monitora costantemente le modifiche apportate alle tabelle di instradamento durante ogni ciclo di elaborazione.

Questa implementazione presenta diversi vantaggi:

- **Ottimizzazione delle prestazioni**: Elimina cicli di calcolo superflui
- **Terminazione automatica**: Il sistema si arresta autonomamente al raggiungimento della stabilità
- **Efficienza computazionale**: Riduce significativamente il carico di elaborazione

Prevenzione di Comportamenti Oscillatori

Un aspetto critico dell'implementazione riguarda la gestione delle situazioni in cui i valori delle distanze potrebbero entrare in oscillazione a causa di percorsi ciclici nella topologia di rete. La strategia implementata prevede:

- **Valutazione comparativa**: Ogni nuovo percorso viene confrontato con quello esistente
- **Aggiornamento condizionale**: Le modifiche vengono applicate esclusivamente quando comportano un miglioramento effettivo
- **Stabilizzazione garantita**: Il meccanismo assicura la convergenza verso una soluzione ottimale

Questo approccio ha eliminato completamente le anomalie comportamentali precedentemente riscontrate nei risultati delle simulazioni.

Validazione e Verifica del Sistema

Per garantire l'affidabilità e la correttezza dell'implementazione, è stata condotta una serie di test approfonditi, tra cui:

Test di Complessità Graduata:

- Topologie semplici con pochi nodi per la verifica base
- Reti di media complessità per testare le prestazioni
- Configurazioni complesse per validare la robustezza

Scenari di Stress Testing:

- Nodi completamente isolati dalla rete
- Collegamenti con costi asimmetrici o molto elevati
- Topologie con percorsi ridondanti multipli

Risultati della Validazione: Attraverso questi test sistematici è stato possibile identificare e risolvere diverse problematiche, in particolare:

- Errori nel calcolo delle distanze verso destinazioni non adiacenti
- Comportamenti imprevisti con valori di costo estremi
- Problemi di gestione delle eccezioni in configurazioni particolari

La metodologia di testing adottata ha garantito un livello di affidabilità elevato del sistema finale, confermando la correttezza dell'algoritmo implementato in tutti gli scenari testati.