Introduzione

Ho svolto la seconda traccia, consistente nella simulazione di un protocollo di routing. In particolare, ho deciso di simulare l'algoritmo Distance Vector (DV).

Struttura e funzionamento

Router

Il file `router.py` contiene la rappresentazione di un router capace di implementare DV.

Metodi

- `add_neighbor`: aggiunge un collegamento diretto con un altro router
- `send`: invia le proprie informazioni di routing ai vicini:

```
for neighbor in self.neighbors:
    neighbor.recv(self, {destination : route for destination,route in self.paths.items()
        if route[0] != neighbor and destination != neighbor})
```

`route[0] != neighbor` evita che vengano inviate al vicino informazioni su destinazioni per le quali esso stesso rappresenta un gateway (split horizon)

- `recv`: riceve le informazioni di routing da un vicino, salvandole in un buffer
- `update`: aggiorna le rotte elaborando le informazioni di routing salvate:

```
for sender,paths in self.received_paths.items():
    for destination,route in paths.items():
        assert(destination != self)
        neighbor_gateway,distance = route
        distance_from_sender = self.paths[sender][1]
        gateway = self.paths[sender][0]
        new_distance = distance_from_sender + distance
        if destination not in self.paths:
            self.paths[destination] = [gateway,new_distance]
        else:
            current_distance = self.paths[destination][1]
            if current_distance > new_distance:
                self.paths[destination] = [gateway,new_distance]
self.received_paths = {}
```

Vengono elaborati tutte le informazioni presenti nel buffer (`received_paths`) a partire dal mittente (`sender`), il quale è un vicino. per ogni percorso (`paths`), se la destinazione è sconosciuta viene inserita nella routing table, altrimenti viene confrontata la distanza attuale (`current_distance`) con la distanza che si avrebbe passando dal mittente, cioè la distanza dal mittente (`distance_from_sender`) + la distanza tra il mittente e la destinazione (`distance`). `neighbor_gateway` è

il gateway tramite il quale `sender` raggiunge la destinazione ma a me in questo contesto non interessa (potrebbe essere utile salvarselo in quanto in caso di modifiche alla tipologia della rete o all'operatività di questo gateway potrei valutare altri percorsi (se ad esempio A può arrivare a F con A->B->D->F, A->C->D->F e A->E->F (in ordine crescente di distanza), in caso venissi a conoscenza della mancata operatività di/modifica delle distanze verso D, avrei le informazioni per valutare di scartare i primi due percorsi e utilizzare il terzo)). In caso persistano ancora dubbi, il codice è ben documentato (ho dovuto rimuovere i commenti per problemi di formattazione).

Attributi

I campi dei router sono così organizzati:

- `name`: stringa contenente un nome identificativo arbitrario (potrebbe rappresentare il MAC address)
- `address`: stringa contenente l'indirizzo IP (fittizio, in quanto si tratta di una simulazione)
- `neighbors`: lista contenente i router vicini
- `paths`: coppie chiavi-valori rappresentante i percorsi e quindi la tabella di routing; la chiave è il router di destinazione, il valore è una lista di 2 elementi: il primo è il gateway (vicino da contattare per raggiungere il router destinazione), il secondo la distanza stimata
- `received_paths`: buffer contenente le informazioni di routing ricevute ma non ancora processate; la struttura è la stessa di `paths`

Main

`main.py` contiene il codice per inizializzare i router e i loro collegamenti e per eseguire la simulazione vera e propria. Si può lanciare con

```
python main.py
```

Setup

La parte di setup consiste nell'inizializzazione dei vari router e nella creazione dei collegamenti. La funzione `create_link()` si occupa di creare un collegamento (bidirezionale) tra due router con una certa distanza.

Simulazione

La simulazione è divisa in round (il numero di essi è il numero di router, cioè il limite superiore) ed ognuno è a sua volta composto di 3 fasi:

- output delle tabelle di routing
- scambio dei percorsi (comunicazione simulata)
- aggiornamento delle rotte

Viene offerta la possibilità di eseguire la simulazione manualmente tramite il flag `--manual`, altrimenti di default la simulazione viene svolta completamente in automatico (`--auto`).

Considerazioni aggiuntive

- Ho utilizzato lo split horizon per risolvere parte dei problemi noti del distance vector (vedi `send()`)
- Ho scelto di simulare la comunicazione tra i router utilizzando un buffer temporaneo come coda dei messaggi perchè volevo mantenere uno stato del sistema che fosse indipendente dall'ordine in cui vengono simulati gli invii dei percorsi (ad ogni "round" tutti i router inviano i percorsi ai vicini ed aggiornano le rotte solo dopo aver a loro volta ricevuto tutto)
- I collegamenti tra i router sono bidirezionali ma il programma è facilmente modificabile per permettere collegamenti unidirezionali
- La configurazione dei nodi non è effettuabile all'interno della simulazione ma ho messo a disposizione, in `main.py`, altre 2 configurazioni
- Durante la realizzazione del programma mi sono accorto e ho corretto la gestione del gateway durante l'aggiornamento della rotte, ovvero il fatto che il gateway da utilizzare non è necessariamente il mittente ma il vicino da utilizzare per raggiungerlo, anche se sia il mittente che il gateway verso il mittente sono dei vicini. Il problema era il seguente: A dista 1 da B e 10 da C, B dista da 1 da C e C dista 1 da D; senza questo accorgimento, il percorso di A per arrivare a D sarebbe stato A->C->D $(10 + 1 = 11)$ al posto che A->B->C->D $(1 + 1 + 1 = 3)$