02/06/2023 – Davide Giannubilo 10775105 & Salvatore G. Karra 10867052

CHALLANGE 3

SOMMARIO

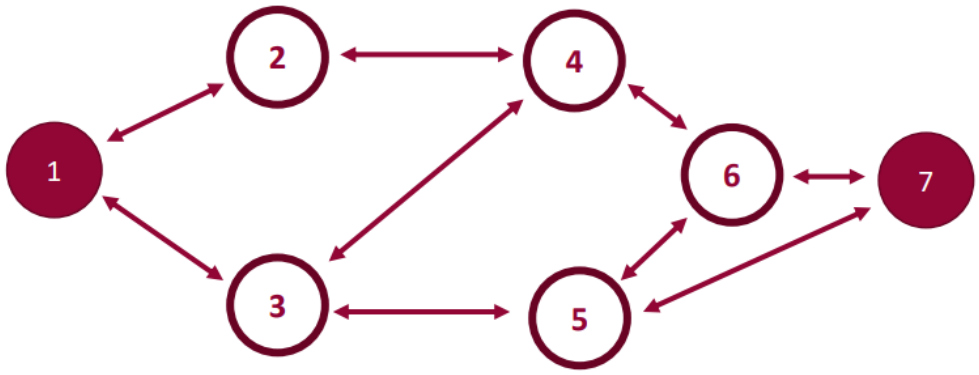
LED STATUS HISTORY FOR NODE 6:
000, 010, 110, 100, 110, 111

TOPOLOGY.TXT

File che abbiamo aggiornato con la topologia indicata nella challenge:

```
1 2 -60.0 (2 è nel raggio di 1 e quindi può comunicarci)
2 1 -60.0 (viceversa 1 è nel raggio di 2 e quindi può comunicarci a sua volta)
1 3 -60.0 (3 è nel raggio di 1 e quindi può comunicarci e viceversa)
3 1 -60.0
2 4 -60.0 (4 è nel raggio di 2 e quindi può comunicarci e viceversa)
4 2 -60.0
3 4 -60.0 (4 è nel raggio di 3 e quindi può comunicarci e viceversa)
4 3 -60.0
3 5 -60.0 (5 è nel raggio di 3 e quindi può comunicarci e viceversa)
5 3 -60.0
4 6 -60.0 (6 è nel raggio di 4 e quindi può comunicarci e viceversa)
6 4 -60.0
5 6 -60.0 (6 è nel raggio di 5 e quindi può comunicarci e viceversa)
6 5 -60.0
5 7 -60.0 (7 è nel raggio di 5 e quindi può comunicarci e viceversa)
7 5 -60.0
6 7 -60.0 (7 è nel raggio di 6 e quindi può comunicarci e viceversa)
7 6 -60.0
```

La topologia creata è quindi la seguente:



RADIOROUTE.H

In questo file abbiamo definito la struttura dei messaggi usati e la struttura della routing table.

```
#ifndef RADIO_ROUTE_H
#define RADIO_ROUTE_H
typedef nx_struct radio_route_msg
{
	nx_uint8_t type; // può essere 0, 1, 2
	nx_uint8_t sender; // chi invia il messaggio
	nx_uint8_t dest; // la destinazione del messaggio
	nx_uint8_t data; //data rappresenta value o cost in base al tipo di messaggio
}radio_route_msg_t;

typedef struct routing_table_entry
{
	uint16_t destAddress; // la destinazione del messaggio
	uint16_t nextHop; // il nodo successivo a quello attuale per raggiungere la
    destinazione
	uint16_t cost; // costo della rotta
}routing_table_entry_t;

enum
{
	AM_RADIO_COUNT_MSG = 10,
};

#endif
```

RADIOROUTEAPPC.NC

In questo file abbiamo inserito le componenti e le interfacce che abbiamo utilizzato.

```
#include "RadioRoute.h"
configuration RadioRouteAppC {}
implementation
{
	/***** COMPONENTS *****/
	components MainC, RadioRouteC as App, LedsC;
	components new AMSenderC(AM_RADIO_COUNT_MSG);
	components new AMReceiverC(AM_RADIO_COUNT_MSG);
	components new TimerMilliC() as Timer0;
	components new TimerMilliC() as Timer1;
	components ActiveMessageC;

	/***** INTERFACES *****/
	App.Boot -> MainC.Boot;
	App.Packet -> AMSenderC;
	App.AMSend -> AMSenderC;
	App.Receive -> AMReceiverC;
	App.AMControl -> ActiveMessageC;
	App.Leds -> LedsC;
	App.Timer0 -> Timer0;
	App.Timer1 -> Timer1;
}
```

RADIOROUTEC.NC

Qui abbiamo il core del nostro programma dove vengono definite tutte le dinamiche per l'invio e la ricezione dei messaggi, per i trigger dei led e per la creazione della routingTable di ogni nodo.

Dopo la definizione del modulo RadioRouteC, delle interfacce e tutte le variabili usate, troviamo diversi eventi e funzioni. Di seguito una breve spiegazione di ognuno di esse.

Funzione: ledSequence()

Questa funzione è stata definita per calcolare i resti della divisione tra il codice persona del leader (10775105) e 3. In questo modo abbiamo creato un vettore contenente i resti.

Funzione: fillRoutingTable()

Questa funzione è stata definita per inizializzare la routing table di ogni nodo. Per farlo l'abbiamo riempita di 0. La struttura risultante è così definita per ogni nodo:

	destAddress	**nextHop**	**Cost**
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0

Funzione: routeTableEmpty()

Questa funzione è stata definita per restituire un valore di tipo bool. In particolare, restituisce TRUE se la prima riga della routingTable è uguale a zero (quindi vuota), FALSE altrimenti.
Se la prima riga della routingTable è vuota, vuol dire che lo sarà tutta la Table.

Funzione: generate_send (uint16_t address, message_t* packet, uint8_t type)

Questa funzione non è stata modificata.

Funzione: actual_send(uint16_t address, message_t* packet)

Questa funzione è stata definita per verificare se un messaggio è stato effettivamente inviato correttamente o se il sistema ha riscontrato un errore. Il controllo utilizza l'interfaccia AMSend.

Evento: Boot.booted()

Avvia il sistema, tramite l'interfaccia AMControl.

Evento: AMControl.startDone(error_t err)

Controlla se la Radio mode del nodo è stata avviata correttamente, altrimenti la riavvia.

Evento: AMControl.stopDone(error_t err)

Stampa un messaggio di stop se viene chiamato.

Evento: Timer0.fired()

Questo evento non è stato modificato.

Evento: Timer1.fired()

Avvia il timer del nodo, imposta la sua routingTable con i valori iniziali definiti sopra. Se il timer è avviato per il nodo 1, possiamo dare inizio al cycle e inviare il primo messaggio in broadcast ai nodi vicini al nodo 1. Impostando il tipo su 1 (broadcast) e destinazione nodo 7

Evento: AMSend.sendDone(message_t* bufPtr, error_t error)

Viene utilizzata quando un messaggio viene inviato e controlla se è stato correttamente e se ci sono stati errori.

Evento: message_t* Receive.receive(message_t* bufPtr, void* payload, uint8_t len)

Qui c'è tutta la logica dell'intero programma, in quanto viene implementata la logica dei led, dell'invio e ricezione di messaggi.

Dividiamo in tre sezioni principali il codice, in quanto, dopo un controllo iniziale sulla lunghezza del messaggio ricevuto, esso si divide in base al tipo di messaggio ricevuto.

- Type → 1 (riga 309)
  Controlla se la destinazione è presente nella routingtable del nodo preso in considerazione.
  1. Se non è presente invia una ROUTE_REQ in broadcast ai suoi vicini.
  2. Se invece il nodo è la destinazione, imposta il costo a 1 (tramite la variabile data) e invia un ROUTE_REPLY in broadcast ai suoi vicini.
  3. Se la destinazione è presente nella routing table, viene inviata una ROUTE_REPLY in broadcast impostando il costo, nella variabile data, come il costo presente nella routing table +1.
- Type → 2 (riga 358)

Anche qui si effettua un controllo sulla presenza del nodo richiesto nella routing table.

1. Se il nodo attuale è il nodo 1 e se ha ricevuto una ROUTE_REPLY con destinazione 7, viene incrementato un contatore e, se il costo di tale percorso è minore di quello che memorizzato, viene sostituito come nuovo percorso migliore. Inoltre, se il contatore ha valore uguale al numero di vicini del nodo 1 (nel nostro caso 2), effettua una stampa a schermo dicendo che è stato trovato un percorso con costo ottimale e invia un DATA MESSAGE al nodo 7 con data uguale a 5.
2. Se la destinazione non è presente, aggiorno la rispettiva routing table e invio una ROUTE_REPLY in broadcast.
3. Se la destinazione è presente nella routing table e, se il costo è minore di quello già presente nella routing table, tale costo viene aggiornato aggiungendo +1 e viene inviata una ROUTE_REPLY in broadcast.

(Il calcolo del percorso ottimo non era richiesto ma ci sembrava giusto implementarlo per seguire la logica generale del programma)

- Type → 0 (riga 419)
    1. Se il nodo 7 riceve un DATA MESSAGE, allora il messaggio è arrivato a destinazione.
    2. Altrimenti, se non è il nodo 7, viene effettuata una ricerca nella routing table affinché si trovi il destinatario del messaggio all'interno di essa. Una volta trovato, viene inviato un DATA MESSAGE al nextHop in modo da seguire il percorso trovato.