Università degli Studi dell'Aquila

Dipartimento di Ingegneria e
Scienze dell'Informazione e Matematica

Report Implementazione WSN di Algoritmi di Localizzazione basati su RSSI

Docenti

Prof. *Luigi Pomante*
Dott. *Marco Santic*

Studenti

Gaetano Fichera Giovanni Lezzi

Anno Accademico 2017-2018

Sommario

1. Fasi di Lavoro

Ci è stato richiesto di progettare ed implementare una rete WSN a supporto di due tecniche di Localizzazione basate su approccio RSSI in un ambiente a due dimensioni.
Il lavoro si è suddiviso in 5 fasi:

1. Studio delle tecnologie da utilizzare;
2. Progettazione dell'architettura in Deployment e Component per la rete WSN;
3. Implementazione su nodi WSN TelosB in nesC avvalendoci di TinyOS;
4. Test dell'architettura in situazioni indoor con raccolta dei dati generati dalla stessa, mediante l'utilizzo di Java;
5. Analisi dei dati raccolti con il supporto di MATLAB.

2. Scenario

Lo scenario considerato prevede l'utilizzo di:

- un numero variabile di *Anchor* che sono nodi la cui posizione è nota e condivisa dagli altri nodi della rete;
- un nodo *Blind* la cui posizione è sconosciuta e deve essere determinata sulla base della posizione delle *Anchor* e il valore del *RSSI* appartenenti ai messaggi condivisi da essi.

Nella fattispecie i nodi sono collocati su di una griglia *2D.*

3. Studio delle tecnologie utilizzate

In una prima fase preliminare ci siamo soffermati sull'approfondimento delle tecnologie da utilizzare, quali:

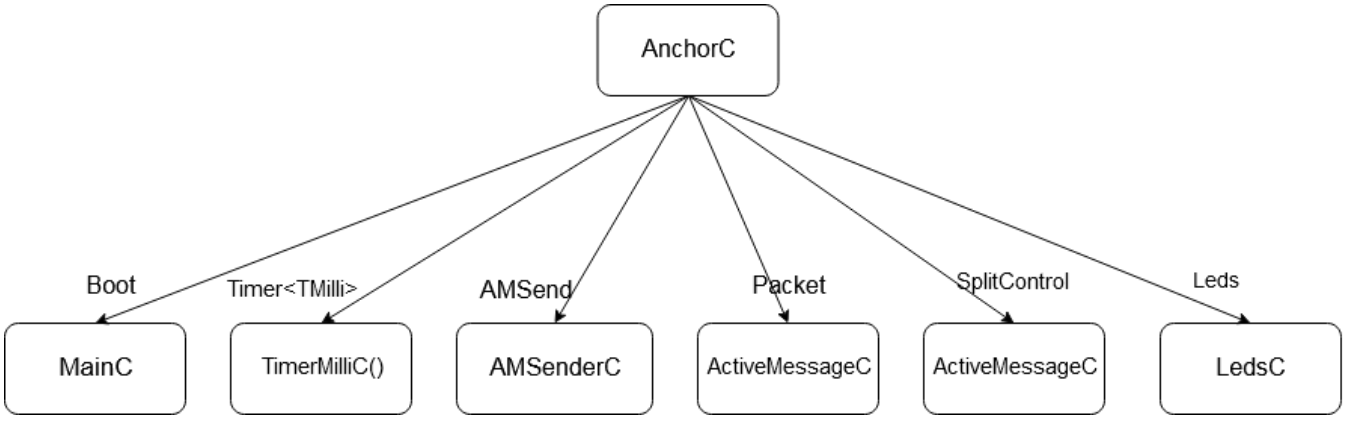
- TinyOS;
- il linguaggio nesC;
- Mote FTDI MTM-CM5000MSP con Chip Radio Texas Instruments C2420, sul quale caricare il nostro codice.

A tal fine ci è stato utile documentarci sulla wiki di TinyOS e in particolar modo il tutorial "RSSI Demo".
Abbiamo quindi implementato una piccola rete con il solo utilizzo di due mote, sui quali effettuare alcuni test per il funzionamento.

4. Progettazione

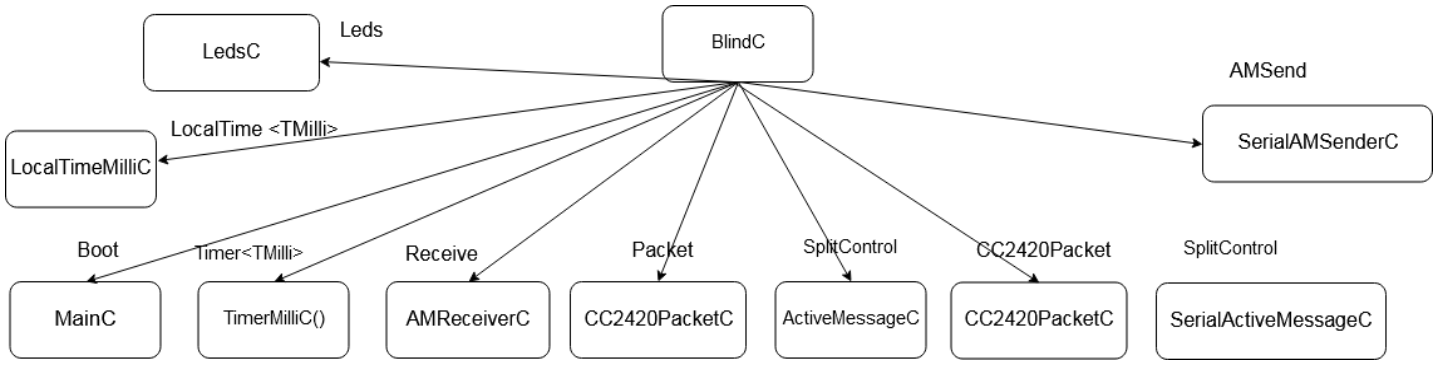
Successivamente ci siamo approcciati al problema che ci è stato sottoposto. Nel progetto vengono identificati due tipi di nodi con distinte funzionalità:

- **Anchor**



  - Identificato da un *ID* (con valori da 1 a N) ed una Posizione definita in Coordinate X e Y;
  - Invio con una determinata frequenza di un *Beacon* via radio per fornire la sua posizione.

- **Blind**



  - Ricezione dei *Beacon*;
  - Estrazione del contenuto dei *Beacon* ricevuti così da rendere disponibile ai due algoritmi di localizzazione, implementati al suo interno, i dati necessari;
  - Computazione dei due algoritmi di localizzazione:
    - *Algo_A:* scansione completa della griglia;
    - *Algo_B:* ricerca indipendete lungo *X0* e *Y0;*

- Invio su seriale dei *Beacon* ricevuti e dell'output degli algoritmi.

Per facilitare tali funzionalità sono state realizzare strutture dati specifiche, in particolare per la trasmissione dei dati abbiamo:

- **BeaconMessage**
  Fornisce il payload da inserire nel *Beacon* che *Anchor* invierà e contiene:
  - *ID* di *Anchor*;
  - Coordinata X e Y di *Anchor*;
  - Periodo in *millisecondi (ms)* dell'invio di *Beacon*.

- **Pos**
  - Coordinata X e Y.

- **Result**
  - Numero di iterazione degli algoritmi dall'avvio;
  - *Timestamp* di inizio e fine elaborazione dei due algoritmi;
  - Coordinate X e Y fornite dall'output dei due algoritmi.

Le restanti strutture dati verranno esposte nei capitoli di *Implementazione* e *Test*.

5. Implementazione

L'implementazione del Progetto è così distribuita:

- **Anchor**
- **Blind**
- **java**
- **matlab**

  BeaconMessage.h
  BeaconRec.h
  Pos.h
  Result.h

5.1. Strutture Dati

Le Strutture Dati, ovvero **BeaconMessage**, **Pos** e **Result,** sono implementate rispettivamente nei file *BeaconMessage.h*, *Pos.h* e *Result.h.* Di *BeaconRec.h* se ne parlerà nel capitolo di *Test*.
Per ognuno di essi, all'interno dei ***.h,*** identifichiamo uno specifico *Active Message Type* a cui verrà associato il messaggio per essere riconosciuto.

5.2. Anchor

All'interno della cartella *Anchor* troviamo:

- **Anchor**

  AnchorAppC.nc
  AnchorC.nc
  ApplicationDefinitions.h
  Makefile

Nel file ***ApplicationDefinitions.h*** vengono raccolti i seguenti dati:

```
#ifndef APPLICATIONDEFINITIONS_H__
#define APPLICATIONDEFINITIONS_H__

enum {
  SEND_INTERVAL_MS = 250,
```

```
  ANCHOR_ID = 1,
  COORDINATE_X = 0,
  COORDINATE_Y = 0
};

#endif //APPLICATIONDEFINITIONS_H__
```

Questi sono utili per memorizzare l'intervallo di tempo con cui vengono mandati i *Beacon*, *l'ID* di *Anchor* e le sue Coordinate X e Y.

In ***AnchorAppC.nc*** troviamo la dichiarazione delle componenti utilizzate da Anchor e l'*include* della struttura dati *BeaconMessage*.

```
#include "BeaconMessage.h"

configuration AnchorAppC {
} implementation {
    components ActiveMessageC, MainC, LedsC;
    components new AMSenderC(AM_BEACON_MSG) as BeaconMsgSender;
    components new TimerMilliC() as SendTimer;
    components AnchorC as App;

    App.Boot -> MainC;
    App.SendTimer -> SendTimer;
    App.BeaconMsgSend -> BeaconMsgSender;
    App.RadioControl -> ActiveMessageC;
    App.Packet -> ActiveMessageC;
    App.Leds -> LedsC;
}
```

In ***AnchorC.nc*** troviamo:

1. La dichiarazione delle interfacce da utilizzare:
2. L'implementazione del suo comportamento è definita in tal modo:
    2.1. Nella fase di avvio del dispositivo si esegue lo start di *RadioControl*:

```
event void Boot.booted(){
        call RadioControl.start();
    }
```

    2.2. Una volta avviato *RadioControl* con successom, viene avviato *SendTimer* con periodo pari a *SEND_INTERVAL_MS*:

```
event void RadioControl.startDone(error_t result){
        call SendTimer.startPeriodic(SEND_INTERVAL_MS);
    }
```

    2.3. All'interno di *SendTimer.fired* si trova il codice relativo all'invio del Beacon:

```
event void SendTimer.fired(){
        Beacon_msg* beaconMsg = (Beacon_msg*) (call Packet.getPayload(&msg,
sizeof(Beacon_msg)));

        beaconMsg->anchor_id = ANCHOR_ID;
        beaconMsg->coordinate_x = COORDINATE_X;
        beaconMsg->coordinate_y = COORDINATE_Y;
        beaconMsg->beacon_period = SEND_INTERVAL_MS;

        call BeaconMsgSend.send(AM_BROADCAST_ADDR, &msg, sizeof(Beacon_msg));
    }
```

5.3. Blind

L'implementazione relativa a Blind è così organizzata:

- **Blind**
  - *ApplicationDefinitions.h*
  - *BlindAppC.nc*
  - *BlindC.nc*
  - *Makefile*

In ***ApplicationDefinitions.h*** abbiamo:

```
#ifndef APPLICATIONDEFINITIONS_H__
#define APPLICATIONDEFINITIONS_H__

enum {
      MAX_ANCHOR = 4,
      BUFFER_DEPTH = 10,
      STEP_SIZE = 1,
      CALC_POS_INTERVAL_MS = 2000,
      STATE_START_ALGO = 0,
      STATE_END_ALGO = 1,
      ID_ALGO_A = 0,
      ID_ALGO_B = 1,
  };

#endif //APPLICATIONDEFINITIONS_H__
```

- *MAX_ANCHOR*: numero di *Anchor* presenti sulla griglia;
- *BUFFER_DEPTH*: numero di valori *RSSI,* per ogni *Anchor,* da tenere memorizzati;
- *STEP_SIZE*: passo di scansione della griglia;
- *CALC_POS_INTERVAL_MS:* intervallo di avvio dei due algoritmi di localizzazione;
- *STATE_START_ALGO, STATE_END_ALGO:* identificativi per gli stati di inizio e fine computazione degli algoritmi;
- *ID_ALGO_A, ID_ALGO_B:* identificativi per i due algoritmi.

In ***BlindAppC.nc*** troviamo la dichiarazione delle componenti utilizzate da *Blind* e l'*include* delle strutture dati necessarie.

```
#include "BeaconMessage.h"
#include "Pos.h"
#include "Result.h"
#include "BeaconRec.h"

configuration BlindAppC {
} implementation {
    components MainC, LedsC;

    components ActiveMessageC as RadioAM;
    components new AMSenderC(AM_BEACON_MSG) as BeaconMsgSender;
    components new AMReceiverC(AM_BEACON_MSG) as BeaconMsgReceiver;
    components CC2420PacketC as TelosBeaconPacket;

    components SerialActiveMessageC as SerialAM;
    components new SerialAMSenderC(AM_RESULT_T) as SerialMsgSender;
    components new SerialAMSenderC(AM_BEACONREC_T) as SerialBeaconRecSender;

    components new TimerMilliC() as Timer0;

    components LocalTimeMilliC, BlindC as App;

    App.Boot -> MainC;
    App.RadioControl -> RadioAM;
    App.BeaconMsgReceive -> BeaconMsgReceiver;
    App.BeaconMsgSend -> BeaconMsgSender;
    App.BeaconPacket -> TelosBeaconPacket;

    App.SerialControl -> SerialAM;
    App.SerialMsgSend -> SerialMsgSender;
    App.SerialBeaconRecSend -> SerialBeaconRecSender;
    App.SerialPacket -> SerialAM;

    App.CalcPosTimer -> Timer0;
    App.Leds -> LedsC;
    App.LocalTime -> LocalTimeMilliC;
}
```

In ***BlindC.nc*** abbiamo:

1. Dichiarazione dei file da includere, tra cui, oltre le necessarie strutture dati, anche la libreria *math.h* necessaria all'interno dei due algoritmi, e la dichiarazione delle interfacce utili:

```
#include "ApplicationDefinitions.h"
#include "BeaconMessage.h"
#include "Pos.h"
#include "Result.h"
#include "math.h"

module BlindC {
    uses interface Boot;

    uses interface SplitControl as RadioControl;
    uses interface Receive as BeaconMsgReceive;
```

```
    uses interface AMSend as BeaconMsgSend;
    uses interface CC2420Packet as BeaconPacket;

    uses interface SplitControl as SerialControl;
    uses interface AMSend as SerialMsgSend;
    uses interface AMSend as SerialBeaconRecSend;
    uses interface Packet as SerialPacket;

    uses interface Timer<TMilli> as CalcPosTimer;

    uses interface Leds;

    uses interface LocalTime <TMilli>;
}
```

2. Tra le variabili globali più importanti abbiamo:
   2.1. uint32_t LUT[80]: array necessario per la linearizzazione dei valori RSSI;
   2.2. *Pos_t posAnchors[MAX_ANCHOR]:* array delle posizioni delle *Anchor;*
   2.3. *bool foundedAnchors[MAX_ANCHOR]:* tiene traccia delle *Anchor* scoperte;
   2.4. *uint16_t buffer[MAX_ANCHOR][BUFFER_DEPTH]:* Matrice contente i valori RSSI dei *Beacon* ricevuti da ogni *Anchor;*
   2.5. *uint8_t bufferIndex[MAX_ANCHOR]:* per tenere traccia dell'ultima Colonna della matrice *buffer,* relativa, riempita;
   2.6. *bool bufferReady[MAX_ANCHOR]:* l' i-esimo valore sarà *TRUE* quando verrà riempita l'ultima colonna della i-esima riga della matrice *buffer;*

3. Il comportamento è implementato in tal modo:

   3.1. *Inizializzazione delle variabili*

```
event void Boot.booted(){
        init();
        call RadioControl.start();
        call SerialControl.start();
    }
```

Quando il nodo *Blind* viene acceso, l'evento *Boot.booted()* inizializza le variabili globali, mediante la funzione *init()* e vengono richiamati i comandi *RadioControl.start() e SerialControl.start()* che sono rispettivamente responsabili del controllo della comunicazione radio e seriale. Se i comandi restituiscono il valore *SUCCESS* allora vengono invocati gli eventi *RadioControl.startDone()* e *SerialControl.startDone();*

   3.2. *Ricezione di Beacon_msg*

```
event message_t* BeaconMsgReceive.receive(message_t* msg, void* payload, uint8_t len) {
        if (len == sizeof(Beacon_msg)) {
                Beacon_msg* beaconMsg = (Beacon_msg*) payload;
```

```
                uint16_t rssi = getRssi(msg);

                handleBeacon(beaconMsg, rssi);
        }
        return msg;
    }
```

Se il controllo radio è positivo verrà richiamato l'evento BeaconMsgReceive.receive() ogni qualvolta un messaggio radio di tipo *Beacon_msg* venga ricevuto. A questo punto dal *Beacon* viene estratto il valore RSSI e, assieme al *payload* del messaggio, viene passato alla funzione *handleBeacon()* che si occuperà di estrarre i valori contenuti nel payload del messaggio.

3.3. *Ricerca delle coordinate di Minimo e Massimo*

```
void handleBeacon(Beacon_msg* beaconMsg, int8_t rssi){
        BeaconRec_t beaconRec;
        uint8_t idAnchor = beaconMsg->anchor_id;

        ...

        idAnchor = idAnchor - 1;

        if (!foundedAnchors[idAnchor]){
                Pos_t posAnchor;
                posAnchor.coordinate_x = beaconMsg->coordinate_x;
                posAnchor.coordinate_y = beaconMsg->coordinate_y;
                posAnchors[idAnchor] = posAnchor;

                calcMinMaxGrid(posAnchor);

                foundedAnchors[idAnchor] = TRUE;
        }

        addToBuffer(idAnchor, rssi);

        ...

       }
```

Per avere gli indici di gestione di array e matrici coerenti si decrementa di uno il valore dell' *ID* dell' *Anchor* che viene ricevuto. Per ogni *Beacon* ricevuto da una nuova *Anchor* la funzione *calcMinMaxGrid(Pos_t posAnchor)* aggiorna gli estremi della griglia confrontando le coordinate di *Anchor* appena ricevute e quelle precedentemente calcolate. In questo modo viene definito un quadrilatero rappresentante la griglia del nostro scenario reale e le posizioni vengono salvate nell' Array *posAnchors;*

3.4. *Riempimento buffer RSSI*

```
void addToBuffer(uint8_t idAnchor, int8_t rssi){
```

```
        uint8_t row = idAnchor;
        uint8_t column = bufferIndex[idAnchor];
        buffer[row][column] = rssi + offsetRSSI;

        increaseBufferIndex(idAnchor);
      }
```

Sempre per ogni *Beacon* ricevuto viene avviata la funzione *addToBuffer(uint8_t idAnchor, int8_t rssi),* la quale salva il valore del RSSI (opportunamente incrementato di un *offsetRSSI*) ricevuto nella matrice *buffer.*

```
void increaseBufferIndex(uint8_t id){
        bufferIndex[id] = bufferIndex[id] + 1;
        if (bufferIndex[id] == BUFFER_DEPTH){
                bufferIndex[id] = 0;
                bufferReady[id] = TRUE;
        }
      }
```

*O*gni riga del *buffer* viene gestita in modo circolare dalla funzione *increaseBufferIndex(uint8_t id)* che si preoccupa, inoltre, di notificare il riempimento di ogni riga del *buffer.* Quando l'i-esimo *indexBuffer* raggiunge il valore di *BUFFER_DEPTH* qusto viene resettato e la corrispondente riga di *bufferReady* viene impostata a *TRUE;*

3.5. *Controllo avvio Algoritmi*

```
void handleBeacon(Beacon_msg* beaconMsg, int8_t rssi){

        ...

        if (!calcPosStarted){
                if (IsBufferReady()){
                        calcPosStarted = TRUE;
                        call CalcPosTimer.startPeriodic(CALC_POS_INTERVAL_MS);
                }
        }
      }
```

```
bool IsBufferReady(){
        uint8_t i;
        for (i = 0; i < MAX_ANCHOR; ++i){
                if(!bufferReady[i]) return FALSE;
        }
        return TRUE;
      }
```

Mediante la funzione *isBufferReady(),* quando il *buffer* è pieno, viene richiamato *CalcPosTimer.startPeriodic(CALC_POS_INTERVAL_MS)*, una volta soddisfatto questo

controllo non verrà più effettuato poichè il *buffer* sarà sempre pieno da quel momento in poi.

```
event void CalcPosTimer.fired() {
        uint8_t i, j;
               for (i = 0; i<MAX_ANCHOR; i++)
                      for (j = 0; j<BUFFER_DEPTH; j++)
                                    bufferCopy[i][j] = buffer[i][j];
        post calcPosTask();
        }
```

Con intervallo *CALC_POS_INTERVAL_MS* viene avviato *CalcPosTimer.fired()* che si preoccuperà di fare una copia del *buffer,* poichè per la natura dinamica di tali dati la stima degli *Algoritmi* su una copia non statica del buffer potrebbe portare a risultati errati. Infine viene avviato il *Task* per il calcolo degli *Algoritmi*;

3.6. *Avvio Esecuzione dei due Algoritmi di localizzazione*

```
task void calcPosTask(){
        Result_t result;

        call Leds.led2On();

        iterAlgo++;
        result.iterazione = iterAlgo;

        result.timestamp_inizio_A = call LocalTime.get();
        AlgoA();
        result.timestamp_fine_A = call LocalTime.get();

        result.coordinate_x_A = posBlindA.coordinate_x;
        result.coordinate_y_A = posBlindA.coordinate_y;

        result.timestamp_inizio_B = call LocalTime.get();
        AlgoB();
        result.timestamp_fine_B = call LocalTime.get();

        result.coordinate_x_B = posBlindB.coordinate_x;
        result.coordinate_y_B = posBlindB.coordinate_y;

        sendToSerial(result);

        call Leds.led2Off();
        }
```

In maniera sequenziale vengono avviati i due algoritmi e popolato un *Result_t* richiamando anche la funzione *LocalTime.get()* per i TimeStamp. Terminata l'esecuzione degli *Algoritmi* la funzione *sendToSerial(Result_t result)* invia il risultato al canale seriale.

6. Test

Per questa fase ci siamo avvalsi di due strumenti: di un programma scritto in linguaggio *Java* e di un programma scritto in MATLAB, utili rispettivamente per la raccolta e l'analisi dei dati. Necessitando anche del riempimento di un file di log contenente tutti i *Beacon* ricevuti dal nodo *Blind* abbiamo utilizzato un'altra struttura dati *BeaconRec.h* contenente una copia del *Beacon* reale con l'aggiunta del suo valore RSSI, estratto in fase di elaborazione dal nodo *Blind,* e il *timestamp* di ricezione.

```
void handleBeacon(Beacon_msg* beaconMsg, int8_t rssi){

        ...

        beaconRec.idAnchor = idAnchor;
        beaconRec.timestamp =  call LocalTime.get();
        beaconRec.rssi = rssi;
        beaconRec.coordinate_x = beaconMsg->coordinate_x;
        beaconRec.coordinate_y = beaconMsg->coordinate_y;
        sendBeaconRecToSerial(beaconRec);

        ...

        }
```

Nel *Blind*, il popolamento e l'invio della copia del Beacon ricevuto avviene all'interno della funzione *hanldeBeacon(Beacon_msg* beaconMsg, int8_t rssi);*

6.1. Java

Facendo riferimento al tutorial, sulla wiki di TinyOs, "*Rssi Demo*" abbiamo due classi java: *BeaconRec* e *Result*, che rappresentano le due strutture dati omonime utilizzate all'interno di *Blind*. Queste sono state ottenute mediante l'utilizzo di *"mig"*, un tool messo a disposizione da TinyOs, attraverso questi due comandi:

```
> mig java -target=null -java-classname=BeaconRec BeaconRec.h BeaconRec_t -o
BeaconRec.java $@

> mig java -target=null -java-classname=Result Result.h Result_t -o Result.java $@
```

Con *BlindMain.java* vengono ricevuti i due tipi di messaggi dalla seriale e stampati su due file di testo con differente struttura:

- *LogBlindAlgo.txt:*

iterazione	*id_Algo*	*Stato_Algo*	*Timestamp Inizio/fine*	*X_Blind*	*Y_Blind*

Nel caso in cui sia uno stato di inizio le coordinate saranno (0,0);

- *LogBlindBeaconRec.txt*

ID_Anchor	*Timestamp Ricezione Beacon*	*RSSI*	*Timestamp*	*X_Anchor*	*Y_Anchor*

6.2. Matlab

Con Matlab abbiamo rielaborato i dati raccolti nei file Log per essere pronti per l'analisi. Inoltre abbiamo implementato in MATLAB i due algoritmi così, utilizzando i *Beacon* raccolti, si è potuto avere un confronto con i risultati raccolti da *Blind*.

All'interno della cartella *matlab* troviamo:

- **fileLogs**
  - **4tests_10x10_step_1_casa_gio_ingiro_18_03_19**
  - **4tests_10x10_step_1_casa_gio_sulleancore_18_03_19**
  - **4tests_10x10_step_1_uni_18_03_14**
- **functions**
  - *AlgoA.m*
  - *AlgoB.m*
  - *avgRSSI.m*
  - *controlloLogBlind.m*
  - *populateConfronto.m*
  - *populateMatlabAlgoA.m*
  - *populateMatlabAlgoB.m*
  - *ControlloMultiploLogBlind.m*

Lo script principale è ***ControlloMultiploLogBlind.m*** che si avvale delle funzioni contenute nella cartella *functions* per effettuare il calcolo relativo ai file Log di 3 test presenti nella cartella *fileLogs*, presentando in output, per ogni test, una matrice per ogni iterazione di ambo gli algoritmi che compara i risultati elaborati in *Matlab* con quelli ottenuti da *Blind*

Nella cartella *functions* abbiamo:

- *AlgoA.m e AlgoB.m* che implementano i due algoritmi presenti in *Blind;*
- *avgRSSI.m* si occupa di ricostruire, conoscendo il timestamp di inizio esecuzione degli algoritmi presente in *LogBlindAlgo.txt* , il buffer utilizzato e di calcolare la media dei suoi valori RSSI presenti in *LogBlindBeaconRec.txt;*
- *controlloLogBlind.m:* si occupa di leggere i due file Log di un singolo test ed elaborare i dati estrapolati per poi compararli;
- *populateMatlabAlgoA.m, populateMatlabAlgoB.m* si occupano di riempire le varie matrici utilizzate successivamente dai due algoritmi come dati di input;
- *populateConfronto.m* si occupa di riempire le matrici di confronto dei vari test effettuati con gli *Algo* implementati in Matlab e in nesC.

Nella cartella *fileLogs* abbiamo raccolto i file Log di tre test da noi effettuati. In tutti e tre i test abbiamo utilizzato:

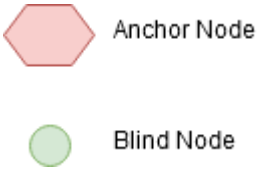
- una griglia 10 x 10 con dimensione reale di 5m x 5m;
- #4 Anchor posizionate sui quattro estremi della griglia e identificate da 1 a 4;

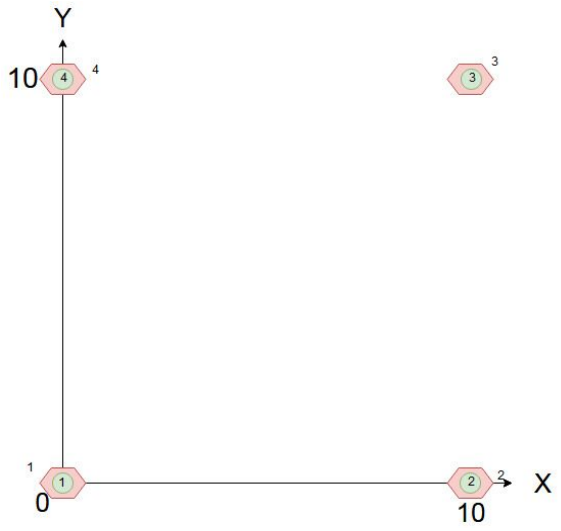
ID Anchor	Posizione
1	0;0
2	10;0
3	10;10
4	0;10

- un intervallo di invio dei *Beacon* per le *Anchor*, *SEND_INTERVAL_MS,* pari a 250ms;
- #1 Blind;
- uno step di scansione della griglia pari a 1;

Tutti i nodi WSN utilizzati sono FTDI MTM-CM5000MSP con Chip Radio Texas Instruments C2420.
Ogni volta abbiamo lasciato eseguire gli algoritmi per uno numero circa di 30 volte su 4 posizioni differenti.

1. *Test 1*





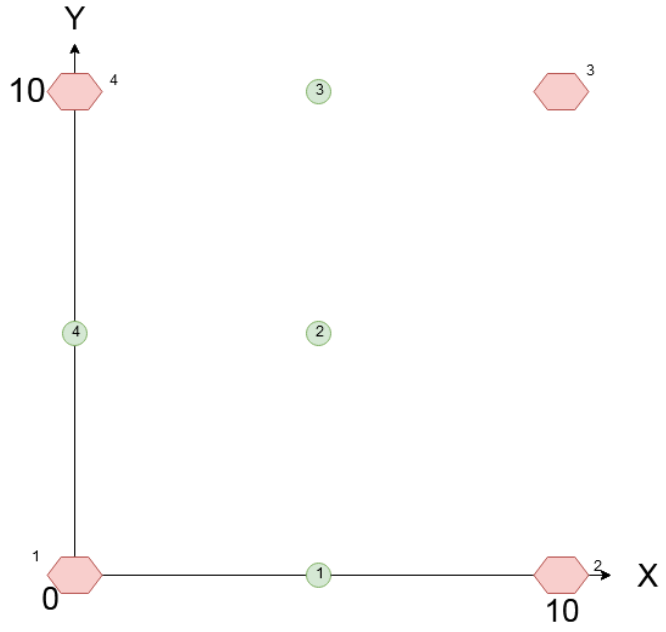
# Prova	Posizione
1	0;0
2	10;0
3	10;10
4	0;10

Il primo Test effettuato è contenuto nella cartella **4tests_10x10_step_1_uni_18_03_14,** qui ci trovavamo in un'aula dell'università, dove tra le *Anchor* si interponevamo soltanto alcuni banchi. Abbiamo posizionato il *Blind* in quattro prove separate sulle quattro posizioni occupate dalle *Anchor.*

2. *Test 2*

Il secondo Test è la cartella **4tests_10x10_step_1_casa_gio_sulleancore_18_03_19,** qui ci trovavamo dentro casa ed abbiamo effettuato le medesime prove del Test precedente.

3. *Test 3*



# Prova	Posizione
1	5;0
2	5;5
3	5;10
4	0;5

L’ultimo Test si trova nella cartella **4tests_10x10_step_1_casa_gio_ingiro_18_03_19**
dove sempre nella stessa location del Test 2 abbiamo variato le posizioni occupate da Blind.

7. Analisi Dati Raccolti

Dalla rielaborazione dei dati raccolti, mediante Matlab, abbiamo potuto constatare che la l'architettura da noi progettata funziona ottimamente ma i risultati ottenuti dagli output dei due algoritmi non sono quelli desiderati. Anche se nei Test 1 e 2 otteniamo valori più o meno esatti, nel terzo Test i risultati sono completamente errati. Al momento non sappiamo quale siano le cause di tali errori.