INDICE

Capitolo 1.Le Reti di Calcolatorie e il modello TCP/IP.. 2

1.1 - Il modello ISO/OSI... 2

1.1.2 - Elenco e funzioni dei livelli OSI.. 3

1.2 - L´Architettura TCP/IP.. 6

1.2.1 - Comparazione tra modello OSI e modello TCP/IP.................................... 7

1.2.2 - Il protocollo di Rete IP... 7

1.2.3 - Caratteristiche del protocollo Ipv6.. 10

1.2.3.1 - Formato dell`Header.. 12

1.2.3.2 - Evoluzione di ICMP (ICMPv6)... 14

1.2.3.3 - L`indirizzo Ipv6... 15

Capitolo 2. Rete wireless.. 17

2.1 - Reti WLAN... 18

2.1.1 - Estensioni di Reti Locali.. 19

2.1.2 - Sistemi di trasmissione di Reti Locali a Onde Radio.................................. 20

2.2 - Lo standard 802.11.. 21

2.2.1 - Le Bande ISM.. 22

2.2.2 - Strato Fisico.. 23

2.2.3 - Protocolli MAC.. 25

2.2.4 - Metodi di accesso al medio RTS/CTS.. 28

2.2.5 - Metodi di accesso al Medio CSMA/CA... 29

Capitolo 3. Wireless Sensor Networks **31**

3.1 -Definizione di WSN 31

3.2 -Architettura generale di un sistema WSN 33

3.3 -Sensori: Interfaccia e condizionamento 34

3.4 - Lo standard IEEE 802.15.4 35

3.4.1 - Topologia di Rete 37

3.4.2 - specificazione PHY 38

3.5 - Lo sviluppo di IPv6 over IEEE 802.15.4. 6LoWPAN 40

3.5.1 - Indirizzi in 6LoWPAN 41

3.5.2 - Composizione Frammentazion Header 6LoWPAN 42

3.5.3 - Compressione Header 6LoWPAN 45

3.5.4 - Principali soluzioni commerciali sul 6LoWPAN 46

3.5.4.1 - Introduzione e descrizione di Nodo-Sensore 46

3.5.4.2 - Diverse opzione comerciali 48

3.5.4.3 - Sensinode. La gamma Nano Series 53

3.6 - Principali Applicazioni di WSN e 6LoWPAN 57

Capitolo 4. Caratterizzazione di un Sistema per la misurazione di parametri ambientali interni.. 58

4.1 - Descrizioni dello 6LoWPAN DevKit K210 of Sensinode NanoSeries...... 58

4.2 - Laboro di configurazione dello Stack di Sensinode.................................. 65

4.2.1 - Primo programa. Invio e Recezione dei dati dei sensori tra due Node........ 67

4.2.1.1 - Pseudo codice emisore Programma 1................................ 69

4.2.1.1 - Pseudo codice recettore Programma 1............................... 72

4.2.2 - Secondo programa. Invio e Recezione dei dati tra diversi sensori............. 74

4.2.1.1 - Pseudo codice emisore Programma 2................................ 77

4.2.1.2 - Pseudo codice recettore Programa 2.................................. 80

4.2.1.3 – Discussione dei problemi incontrati.................................. 82

4.3 - Prove realizate nel laboratorio. Copertura Radio....................................... 86

4.3.1 - Prima Prova: Radio di Copertura della rete in spazio coperto..................... 87

4.3.2 - Seconda Prova: Radio di Copertura della rete nel laboratorio..................... 90

Capitolo 5. Allegati.. 94

-Codice Primo programa. Emisore.. 94

-Codice Primo programa. Recettore.. 107

-Codice Secondo programa. Emisore.. 117

-Codice Secondo programa. Recettore.. 130

Escenarios experimentales en una red de sensores inalámbrica con IPv6

(Experimental scenarios using a Ipv6 Wireless Sensor Network)

Introducción

El interés por las redes de sensores inalámbricas o WSN (*Wireless Sensor Networks*) ha ido creciendo durante los últimos años. En el campo de la telemática se ha notado un ferviente interés ya que las WSN se pueden utilizar en multitud de aplicaciones telemáticas que permiten la intearactuación con el medio de una forma más directa.

Las redes de sensores inalámbricas o WSN (*Wireless Sensor Network*) son conjuntos de dispositivos de bajo coste, llamados comúnmente *motes*, con una interfaz inalámbrica, que permiten, mediante elementos sensores, recolectar todo tipo de información del mundo físico.

La facilidad y flexibilidad en el despliegue de las WSN es uno de sus principales puntos fuertes, lo que hace de este tipo de redes un candidato idóneo para un gran número de aplicaciones: monitorización de entornos, control de todo tipo de dispositivos y recolección de datos en bruto son las principales aplicaciones de las WSN.

Hoy en día existen numerosas empresas que se dedican a la creación de *motes* que permiten la lectura de datos capturados por los sensores incorporados, ademas de trasmitir y recibir estos datos actuando así como un nodo de comunicaciones independiente. Esta funcionalidad dota a este tipo de redes de un especial interés, ya que obtener datos en tiempo real pero de forma remota.

En la actualidad, una de las labores en las que se están invirtiendo más horas de investigación es la conseguir implementar IPv6 en las redes de sensores inalámbricos, haciendo que sean parte de la nueva generación de Internet, y con esto dotar de una total integración las WSN en la *World Wide Web.* El organismo encargado de la estandarización de IPv6 en redes inalámbricas es el denominado *Internet Engineering Task Force´s IPv6 Low Power Wireless Personal Area Network* (6LoWPAN). Su reto es implementar IPv6 de manera que el protocolo pueda ejecutarse en sensores de dispositivos pequeños, que funcionan generalmente con baterías y que operan en frecuencias de radio reducidas y de bajo consumo energético, en este caso basadas en el estándar IEEE 802.15.4. Cada nodo en una red de este tipo podría convertirse en otro nodo IP, directamente accesible por otro nodo sensor o por otro nodo en otra IP.

El trabajo realizado en el proyecto se ha llevado a cabo con el hardware y entorno de

desarrollo *"6LoWPAN DevKit K210 of Sensinode NanoSeries*", con el que se han elaborado diversos programas para ir experimentando con este modelo en concreto de mote.

El principal objetivo en este proyecto, es desarrollar un software para la transmisión/recepción de datos de una red de sensores inalámbrica, así como realizar mediciones para testar el funcionamiento de este software en el entorno del laboratorio "Eriksson" de la planta H de la facultad de ingeniería de manera que se generen los cimientos para la posterior elaboración de una red de sensores Ipv6 que permita la monitorización de las variables ambientales en la *Università degli studi di Pavia.*

Breve descripción del entorno de trabajo

El "6LoWPAN Devkit k210 de Sensinode" esta compuesto por:

- 1 placa DevBoard D210,
- 4 NanoSensores N711,
- 2 dispositivos NanoRouter N601
- 2 módulos de Antenas radio N100 con el microchip CC2431.

Figura 1. Development kit

El microchip CC2431 también está incluido en los NanoSensores N711, el DevBoard D210 y los NanoRouters N601, puesto que estos tres tipos de dispositivos, tienen incorporado un modulo de radio N100.

Se ha trabajado programando los sensores N711 a través del placa DevBoard D210, testando de esta manera el funcionamiento del software realizado. El modo en el que se disponen los distintos componentes es; la placa D210 se conecta mediante el puerto USB al PC, al mismo tiempo, la

placa D210 permite tanto la lectura de datos desde el Nodo Sensor N711 mediante un puerto UART como la programación de estos dispositivos mediante el puerto PRG del DevBoard 210.

El DevBoard D210 no solo es el encargado de realizar las tareas de programación en los Nodo Sensores N711 sino que ademas hace de enlace para la lectura de datos desde el PC. El nodo Sensor N711 esta compuesto por un modulo de Radio N100 que como se ha dicho anteriormente incluye un microchip CC2431, que ofrece 250Kbps de velocidad de trasmisión. Este es el dispositivo que se programa mediante el puerto PRG a través del DevBoard D210. Ademas, el N711 contiene 2 sensores, uno de Luz, y otro de temperatura, ademas de 2 Leds y 2 botones.

Para la lectura del puerto USB en caso de utilizar se ha optado por la utilización del programa recomendado por Sensinode, Hyperterminal. Este software ha permitido la lectura a tiempo real de los datos que hacíamos enviar al N711 a través del UART. Siendo de esta manera la única forma de comprobar el correcto funcionamiento del software realizado dada la falta de un entorno para emular la ejecución del código. De esta manera, para probar en los sensores el funcionamiento de cada uno de los programas, ha sido necesario, cada vez que se realizaba algún cambio en el código, la reprogramación del sensor.

Otra de las características del entorno seleccionado es la utilización de un emulador POSIX como Cygwin para poder trabajar en Windows con la pila de protocolos de NanoStack. A continuación se muestra la composición interna de la arquitectura de NanoStack:

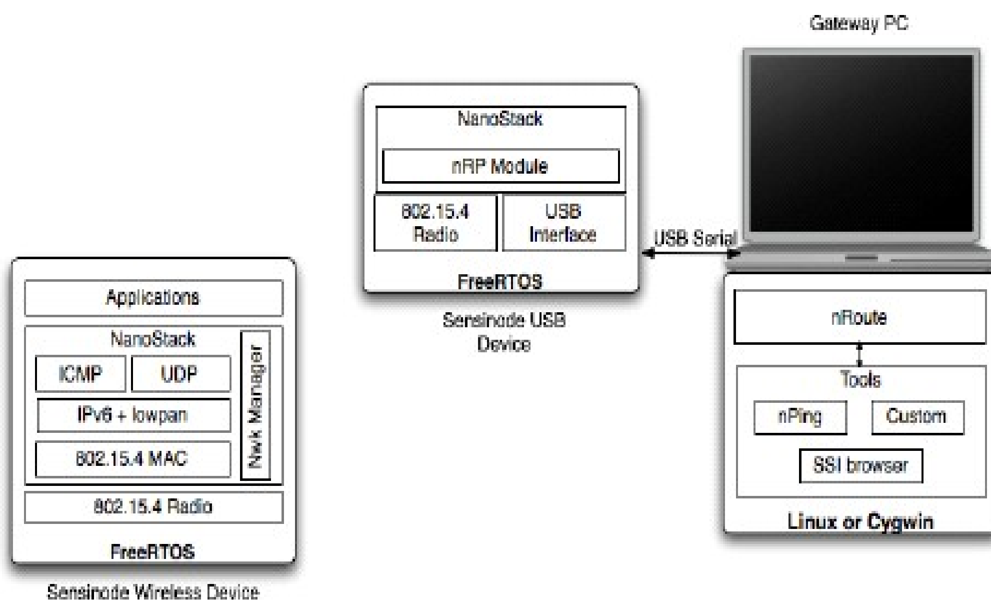


Figura 2. Arquitectura NanoStack

Descripción programa1: envió/ recepción de datos entre 2 nodos

El principal objetivo de este primer programa es la de obtener los datos de Temperatura y Luz tanto del sensor local como de un sensor remoto. El programa esta dividido en un primer código “emisor, y un segundo código “receptor”. La funcionalidad de cada uno de ellos se explica a continuación.

El programa “emisor”:

Este programa se instala en el nodo “sumidero”, es decir, aquel nodo que vaya a estar conectado mediante el UART al DevBoard D210, que a su vez va a estar conectado mediante el puerto USB al PC. El nodo sumidero será el encargado de recolectar toda la información que le llegue del nodo receptor, con el fin de tratar dicha información para enviarla al usuario final (la aplicación).

El programa emisor se encarga de interrogar al nodo receptor, enviando paquetes de consulta en intervalos temporales definidos por el usuario. La consulta finaliza cuando el usuario lo desee o cuando el número de paquetes enviados supere los 50. El programa también presenta otras opciones de funcionamiento:

- Menú de consola para facilitar el manejo del programa por parte del usuario,
- Configuración de potencia de trasmisión del nodo remoto,
- Envió de ping TCP y UDP
- Impresión por pantalla tanto de los resultados de los Pings como de los datos contenidos por los paquetes enviados por el sensor remoto.

El programa “receptor”:

Este programa se instala en el nodo remoto. Va a ser el programa encargado de esperar una petición del nodo “sumidero” para enviarle un paquete donde vengan los datos de Luz y Temperatura. Por cada paquete REQUEST que le llegue, él enviará un paquete de tipo RESPONSE. También incluye, como en el programa “emisor”, la funcionalidad de consola, ping, y potencia. Tanto para ser configurado de forma remota, como para realizar la configuración mediante el puerto PRG a través del DevBoard D210.

Nota: Tras algunas modificaciones, como añadir la funcionalidad de cambiar la potencia de trasmisión, el programa descrito también ha sido utilizado para realizar distintas mediciones de alcance radio en los nano sensores.

```
PROYECTO- DEVBOARD-N100 - HyperTerminal
File  Edit  View  Call  Transfer  Help

             NanoStack Start Ok
    EMISORE UNICAST. TX TEST. Versione 1.0

Open and bind Send s1 socket
Open and bind receive socket
**********
Shell help:
1 - Start listen process
2 - Finish listen process

+ - Power up 10
- - Power down 10
**********

INSERT WAITING TIME in sec. (0,9]:
 2 1000
Mode Sending Request.
REQ
Send OK

-----------

00:00:44:44
Temperature ‖C:  13Luce:  23892


-----------
Request Send:  1 Recived ok:  1
REQ
Send OK
```

Figura 3. Captura de datos del nodo emisor

Descripción programa2: envio y recepción de datos entre varios nodos.

En este el segundo programa, se pretende conseguir la recepción en un nodo central o sumidero, de los datos de todos los sensores que se encuentren dentro de su rango cobertura radio. De la misma manera que en el programa1 el nodo sumidero interroga cada cierto tiempo al nodo remoto, en este caso, el funcionamiento va a ser el mismo, pero teniendo en cuenta que el nodo sumidero va a realizar el envío de un paquete “REQUEST” en modo Broadcast a todos los nodos que se encuentren dentro de su radio de cobertura, estos van a recibir el mensaje y van a trasmitir, en intervalos de tiempo distintos, para evitar colisiones, un único paquete que va a contener la información de los sensores remotos. El nodo sumidero se encargará de mostrar la información especificando de cual de los sensores la ha recibido.

Como en el programa1, también habrá 2 programas; uno “emisor” (para el nodo que ejerce de sumidero), y otro “receptor”, para los nodos remotos de los que se pretende obtener información. En este caso, no es exactamente que vaya a existir un solo programa receptor, ya que para solucionar una serie de problemas que se detallan a continuación, este programa receptor va a ser distinto para

cada uno de los nodos en los que se instala, ya que para cada uno de los nodos se va a especificar un tiempo distinto de trasmisión de los datos, evitando así posibles colisiones entre los nodos receptores.

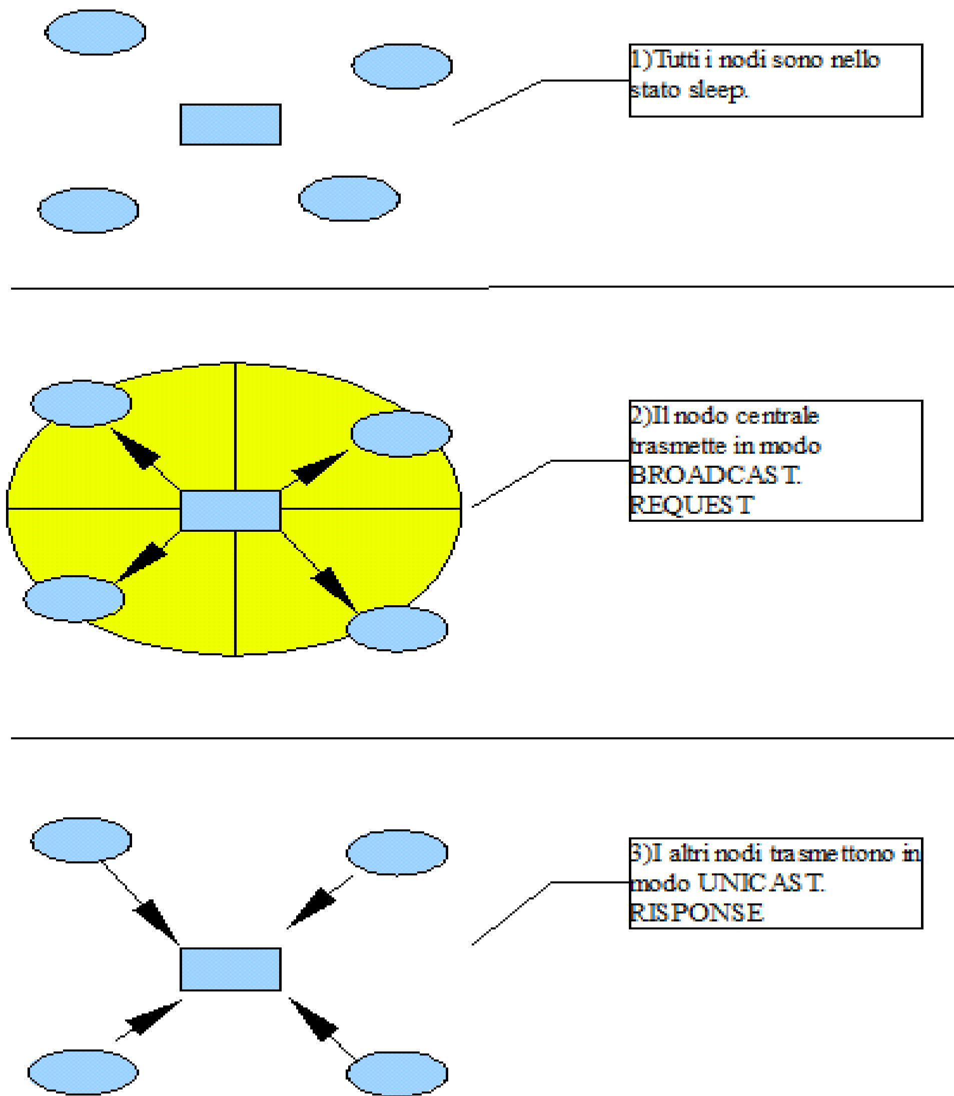


Figura 4. Funcionamiento programa 2

Programa2: problemas encontrados y discusión de los mismos

El programa2, como se ha comentado en la sección anterior, ha sido foco de ciertos problemas que se han intentado solventar mediante diversas soluciones ya que se buscaba alcanzar los objetivos iniciales propuestos para este programa.

El problema consiste en que cuando en la comunicación entre los nodos intervienen mas de 2 nodos, receptor y emisor, y se introducen varios nodos receptores mas, ocurre lo siguiente:

Después de diversas trasmisiones, los nodos receptores, de forma aleatoria van dejando de trasmitir información al nodo central hasta quedar en ultima instancia, solo un nodo conectado al nodo central, enviándole información. Justo en el momento en el que uno de los nodos deja de trasmitir, este, enciende el LED2 del NanoSensor N711, sin seguir el código programado. Ya que en las pruebas realizadas para comprobar exactamente porque sucede esta situación, se eliminaron del código las sentencias que modificaban el estado de los LEDs.

Después de realizar varias pruebas, se descarta que el error sea del código elaborado. Por ello, se decide afrontar el problema. Se plantean los posibles fallos que pueden estar provocando el mal funcionamiento del programa. Se plantean los siguientes casos:

- ¿Existen colisiones?¿En caso de existir, donde se producen?.
- ¿Memoriza correctamente el nodo receptor la dirección del nodo emisor?
- ¿Procesa el nodo central o emisor todos los paquetes?
- ¿Porque se enciende el LED2 del N711, cuando en el código no aparece ninguna sentencia que ejecute esta instrucción?

Para solucionar cuestiones relacionadas con el primer punto y segundo punto, y tras llevar a cabo varias modificaciones en el programa realizado al inicio, se decide generar un programa distinto para cada nodo receptor. Cada nodo transmitirá en un intervalo temporal distinto, evitando así que se produzcan colisiones. Además, los nodos ya no envían el paquete RESPONSE con la dirección obtenida del paquete REQUEST, sino que la dirección del nodo sumidero la conocen *a priori*. De esta manera se descarta que el problema este en la memorización correcta de la dirección del nodo central, por parte de los nodos receptores, ya que estos están programados manualmente con la dirección del nodo al que tienen que enviar. Esta solución simplifica considerablemente el programa, ya que hace que sea mucho menos genérico, y requiera que cada nodo tenga su propio programa. Tras las pruebas pertinentes, los problemas se seguían sucediendo, así que se añadieron de nuevo las funcionalidades omitidas con la convicción de que el problema no se encontraba en el código, si no en algún otro lugar como en el archivo de configuración FreeRTOSConfig.h.

Sobre el tercer posible fallo, en el que el nodo central o emisor sea el que no procesa o recibe bien los paquetes, se llega a la conclusión realizando varias pruebas y observaciones, de que es evidente que el problema no esta aquí, puesto que siempre que el nodo remoto envía bien la información, esta es procesada correctamente por el nodo central o emisor. Luego el problema esta en los nodos remotos emisores, ya que son estos los que dejan de trasmitir correctamente cuando actúan en una red con mas receptores. En las pruebas realizadas con este mismo código, pero solo con 2 nodos, un sumidero o emisor, y un solo receptor , no sucede ningún problema, y en ningún momento el LED2 del receptor se enciende y deja de trasmitir.

Sobre el último posible fallo, el encendido del LED2 del N711, éste se sucedía justo en el momento en el que el NanoSensor dejaba justo de trasmitir. En este momento, al leer por consola el sensor, éste dejaba de ejecutar su código. Simplemente, se paralizaba. Lo primero que se hizo fue comprobar el código y revisar exactamente donde se daba la instrucción al N711 para que encendiera el LED2, y ver si el problema estaba cerca de esta instrucción en el código, puesto que el LED2 quedaba encendido en el tiempo que el N711 quedaba bloqueado. Tras comprobar cada uno

de los lugares en que estaba escrita esta sentencia a través de debugs, se llegó a la conclusión de que el problema no parecía estar relacionado con esta sentencia. Para evitar, y comprobar que efectivamente fuera así, se decidió eliminar toda sentencia que ejecutara el encendido del LED2 en el código. Tras este cambio, seguía sucendiendo lo mismo, es decir, en un momento determinado, cuando actuaban 2 o mas receptores, estos se iban bloqueando, encendiendo, a la vez que se bloqueaban, el LED2. Ante este problema se trabajo después en los puntos anteriormente descritos, para ver si era consecuencia de alguno de ellos, tras esto no se pudo llegar a falta de mas tiempo material, a una conclusión exacta.

Tras no poder resolver el problema que surgió, se decidió, dentro del margen de tiempo del que se disponía, indagar un poco mas sobre la configuración de FreeRTOS en el Stack de Sensinode, y sobre todo de los archivos FreeRTOSConfig.h. Estos archivos son donde se configuran las distintas opciones del Stack, como por ejemplo el tamaño máximo de los buferes, la configuración de la potencia de transmisión y parámetros similares. Para la realización del código, se tomo como base el programa ejemplo “nano_sensor_n71x” de NanoSeries de Sensinode, y por tanto, también se cogió su archivo FreeRTOSConfig.h. Después de realizar algunas pruebas no se logró llegar a una conclusión concreta.

Desde el punto de vista del autor de este proyecto, el problema debe estar localizado en la trasmisión de los datos por parte de los nodos receptores o remotos, o por ultimo, en la configuración de las propiedades del stack de sensinode. Se propone la solución de este problema como una linea de trabajo futuro.

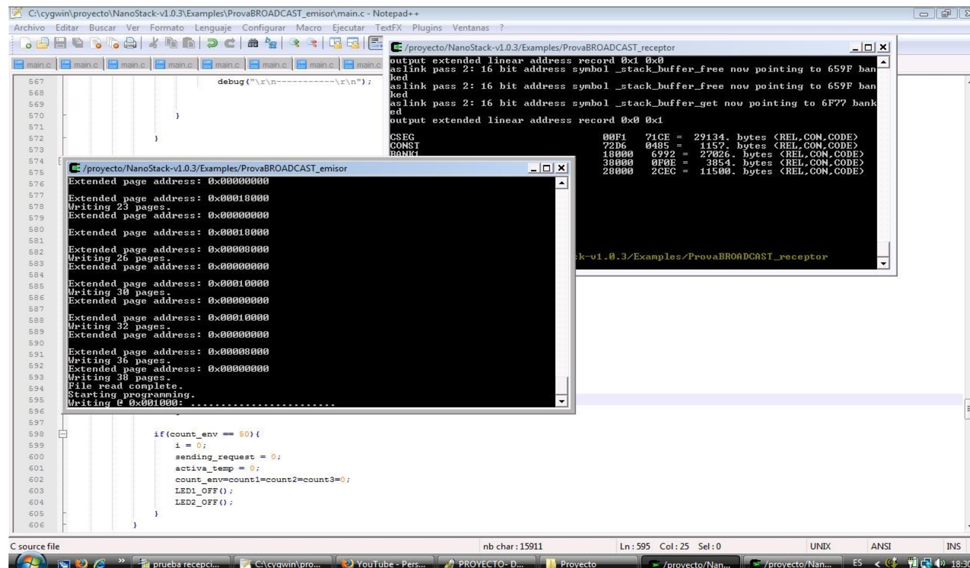


Figura 5. Recepción de datos en nodo sumidero

Resumen de los resultados de las pruebas realizadas en el laboratorio

Las pruebas se desarrollan en el escenario del laboratorio Erikson de la planta H en el edificio de la facultad de ingeniería de la *Università degli Studi di Pavia*. Para las 2 pruebas se ha utilizado una tasa de trasmisión de un paquete por segundo.

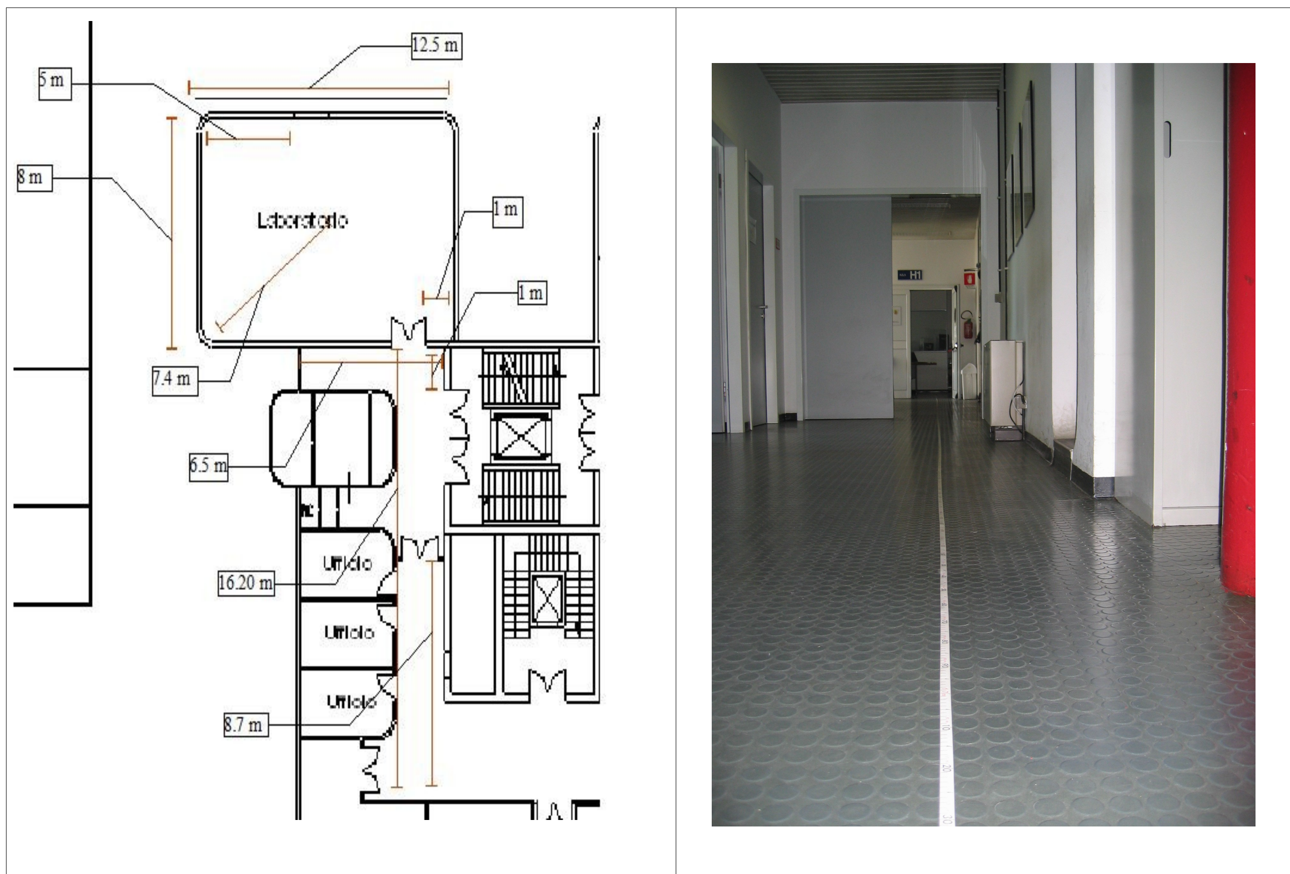


Figura 6. Escenario de pruebas: Planta H del edificio de ingeniería

PRUEBA 1:

La primera prueba pretende comprobar el radio de cobertura de 2 nodos entre si en un espacio cubierto, en linea recta y sin obstáculos. Para ello se utiliza el programa1 (UNICAST) descrito en la anterior sección. Se considerará que un paquete es correcto cuando el nodo central, después de enviar una petición al nodo receptor o remoto, recibe un paquete de respuesta que es procesado correctamente.

Para realizar las medidas se utilizan dos sensores N711, con el programa1 emisor instalado en el nodo1, y el programa1 receptor instalado en el nodo2. Los resultados obtenidos se van a relacionar

con la potencia de trasmisión a la que están configurados los nodos, para así poder contrastar el radio cobertura con la potencia de trasmisión utilizada.

A continuación se muestran los resultados obtenidos:

- Se pueden diferenciar 4 gráficas. Cada una de ellas configura el nodo emisor y receptor a la potencia descrita en el titulo.

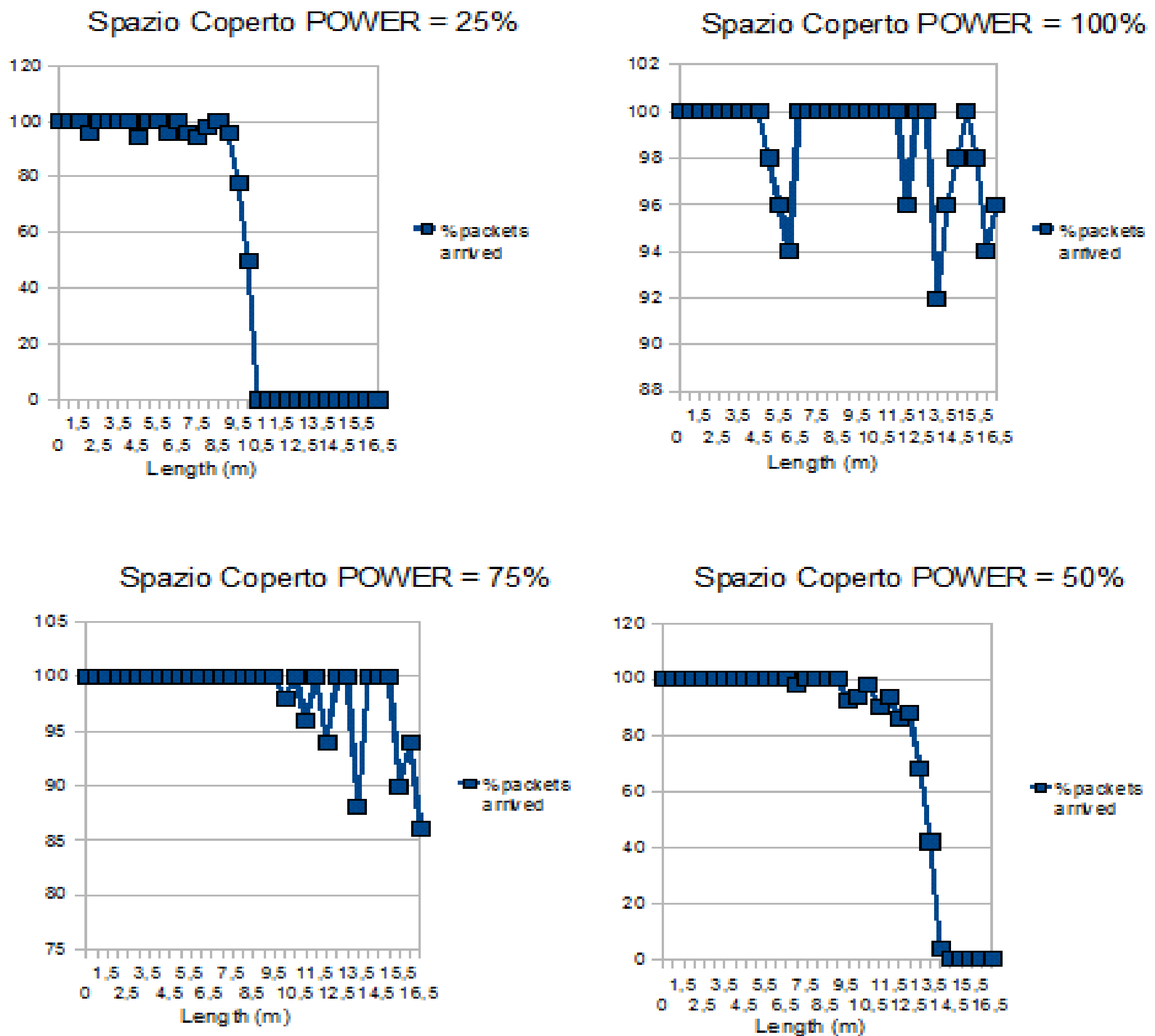


Figura 7. Resultados medidas experimentales potencia

En las gráficas se muestra la relación del % de paquetes recibidos correctamente, es decir, paquetes de los que se extraen correctamente los datos de luz y temperatura, con la distancia a la que llegan los paquetes,. Ademas cada una de las gráficas muestra también la configuración de potencia de los nodos. Cuando en una gráfica se describe un POWER = 100%, quiere decir que tanto el nodo emisor, como el nodo receptor, están ambos configurados para trasmitir a una potencia del 100%.

Los resultados demuestran que, para un espacio cubierto sin obstáculos, los 16.5m del pasillo de la planta son asumibles, si se cumple el objetivo fijado de un rango entre el 80% y el 90% de paquetes recibidos correctamente. Por tanto, configurando la potencia de trasmisión de los nodos a un 75%, se puede observar (queda reflejado en la gráfica "*Spazio coperto POWER = 75%*") que se obtiene un mínimo % de paquetes correctamente recibidos, esto es, una recepción entre el rango del 80% y el 90% de paquetes recibidos correctamente a la máxima distancia posible en el pasillo, en la que se obtiene concrétamente un 86% de paquetes recibidos correctamente. Por tanto se puede asegurar, que configurando al 75% la potencia de trasmisión tanto el nodo emisor, como el nodo receptor, se obtienen unos resultados que cumplen el objetivo mínimo planteado de obtener un 80% y un 90% de paquetes recibidos en los que se ha podido leer correctamente los datos de luz y temperatura.

Para poder analizar bien una solución para la configuración de una WSN 6LoWPAN en el piso H, se opta por realizar otra prueba, donde se incluyen diferentes obstáculos, y escenarios. A esta prueba la llamamos Prueba 2.

PRUEBA 2:

Esta prueba, como la anterior vuelve a utilizar el programa1, ubicándose en el mismo escenario, el pasillo de la planta H. En este caso se realizan tres test diferentes.

- **Test 1:** consiste en ver la influencia de las diferentes puertas de la planta H en la trasmisión de datos. En esta prueba se utiliza una potencia de trasmisión tanto en el nodo central, como en el nodo receptor directamente al 100%, con una distancia total de 23.5m entre un nodo y otro. Se va a modificar el entorno de forma constante, abriendo y cerrando las dos puertas de la planta para ver como influyen los obstáculos en la recepción de los datos.
- **Test 2:** Similar al primer experimento pero manteniendo la puerta del pasillo cerrada para comprobar la influencia de este obstáculo en la recepción de datos, estudiando la variabilidad de la recepción, así como la cantidad de metros de margen que se tienen para buscar una instalación optima de los sensores. Aquí la potencia de trasmisión es también del 100%, pero se van a realizar distintas mediciones según la distancia entre los dos nodos, teniendo en cuenta que la separación máxima será igual a la profundidad del pasillo, es decir 16.5m.
- **Test 3:** Se va a medir el porcentaje de paquetes recibidos correctamente en la sala del propio laboratorio Erikson. Para ello, se decide realizar las mediciones en una hora en la que el laboratorio tiene una afluencia de gente normal, es decir, en horario laboral, con los ordenadores funcionando y la gente trabajando normalmente. Se van a medir las dos diagonales del laboratorio. Cada diagonal mide exactamente 14.8 metros.

A continuación se muestran los resultados:

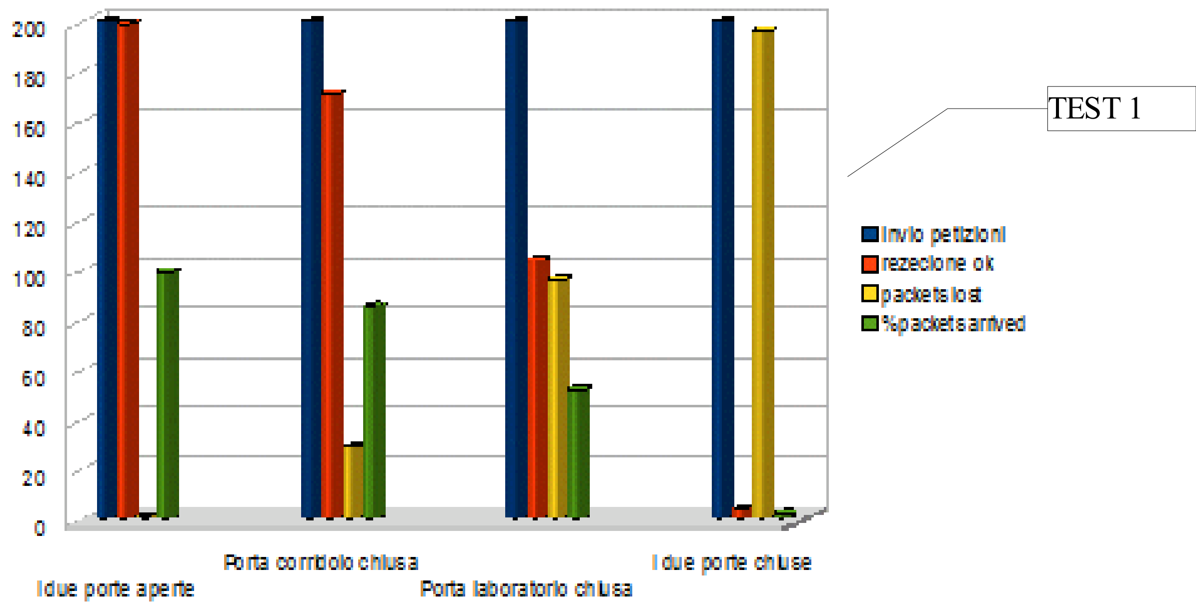
En el Test1, se puede ver como con las 2 puertas abiertas, prácticamente no se obtiene ninguna perdida. Como la puerta del pasillo cerrada genera ciertas perdidas, pero mas atenuación aun provoca la puerta del laboratorio. En el caso de cerrar las 2 puertas, el resultado desde dentro del laboratorio al final del pasillo es completamente nefasto, ya que no llega prácticamente ningún paquete.

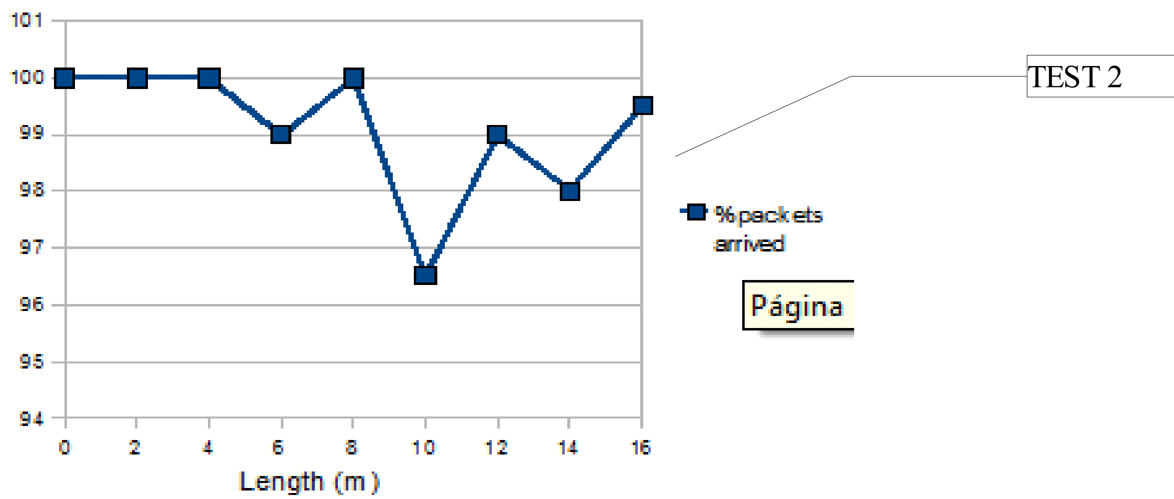
En la gráfica se muestran las situaciones:
-2 puertas abiertas.
-Puerta pasillo cerrada - laboratorio abierta.
-Puerta pasillo abierta - laboratorio cerrada.
-2 puertas cerradas.

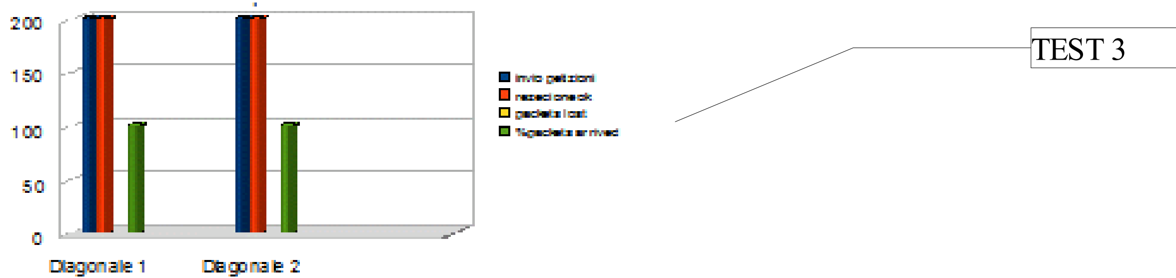
Por cada una de las situaciones se muestran una serie de barras en el gráfico, que muestran: Numero peticiones enviadas, numero de paquetes recibidos correctamente, numero de paquetes perdidos y porcentaje de paquetes recibidos correctamente.

En el Test2, podemos comprobar, que no influye prácticamente esta puerta en la correcta trasmisión y recepción de los datos, de esta manera, y con los resultados obtenidos del test uno, se puede deducir que la puerta que genera mas problemas es la del laboratorio, y por tanto deberá ser un punto clave a la hora del diseño en la correcta colocación de los nodos.

En el Test3 se muestra una gráfica que sigue el mismo esquema que en el test1, pero en este caso se muestran los resultados sobre las 2 diagonales del laboratorio.


200
180
160
140
120
100
80
60
40
20
0
Idue porte aperte
Porta corridoio chiusa
Porta laboratorio chiusa
I due porte chiuse
Invio petizioni
rezecione ok
packets lost
%packets arrived
TEST 1


101
100
99
98
97
96
95
94
0
2
4
6
8
10
12
14
16
Length (m)
%packets arrived
Página
TEST 2


200
150
100
50
0
Diagonale 1
Diagonale 2
TEST 3

Conclusiones y Lineas Futuras.

La incursión de 6LoWPAN en el mundo de las WSN (*Wiereless Sensor Networks*) supone un gran cambio de perspectiva a la hora de valorar las ventajas que una red de sensores inalámbrica puede ofrecer. Conseguir alcanzar el paradigma de la conectividad total a través de IP versión 6, incorporando las redes de sensores como un elemento IP más.

La estandarización 6LoWPAN a permitido todo un desarrollo de nuevos tipos de "motes", motes con capacidades todavía en experimentación, que comienzan a aparecer en versiones comerciales, pero con todavía mucho camino por recorrer, tanto en determinar sus verdaderas capacidades, como en desarrollar software para aprovechar estas cualidades al máximo.

Las pretensiones de este proyecto se encaminan a experimentar con este tipo de motes, para dejar una pequeña base que en el futuro permita "trazar" en la *Università degli studi di Pavia* una linea de estudio encaminada al desarrollo de software libre para el control y desarrollo de aplicaciones en motes 6LoWPAN.

En el trabajo se recogen 2 programas, que aprovechan varias de las funcionalidades de los motes y que son el fruto del trabajo realizado. Estos programas son los que permitirán, a quien afronte el reto de continuar la linea de trabajo aquí iniciada, a no tener que detenerse en funcionalidades básicas como la comunicación unicast y broadcast entre sensores, o la extracción, traducción y empaquetamiento de los datos provenientes de los sensores para su posterior recepción remota en otro nodo.

También deja concluso un trabajo de introducción para el programador de como configurar y programar los motes de la gama de NanoStack de Sensinode, que son con los que se ha trabajado, facilitando la posterior labor del programador, así como realizando una reflexión sobre que forma de trabajo elegir para desarrollar software para con estos motes.

La ultima parte del trabajo es una pequeña estimación del funcionamiento del software realizado con los motes de la gama de nanostack de Sensinode en el entorno de la Planta H de la *Università degli studi di Pavia*. Que tiene como finalidad realizar una aproximación de como va a ser el comportamiento de estos motes en un entorno determinado, en el que a *posteriori* se pretende realizar la implementación de una red de control ambiental, primero en la planta del laboratorio, después en todo el edificio, y finalmente en todo el campus de ingeniería.

Esta red, que es el objetivo en ultimo termino, permitiría a tiempo real desde cualquier punto con un acceso a Internet, conocer medidas de distintas variables ambientales en todo el campus y a partir de estas, corregir y controlar, ya sea de forma automatizada o manual, otra serie de instrumentos electrónicos para una domotización, por ejemplo, de los aires acondicionados, la subida y bajada de persianas, o el encendido y apagado de las luces.

Como se puede ver este proyecto se corresponde tan solo a la primera piedra, en la construcción de un proyecto mucho mas ambicioso y a largo plazo.

Capitolo I. Le Reti di Calcolatorie e il modelo TCP/IP.

1.1 Il modello ISO/OSI.

Il modello di riferimento ISO/OSI (Open System Interconnection) rappresenta lo standard che i costruttori di apparecchiature di rete devono adottare per la gestione delle diverse fasi del processo di trasmissione.

Le tecniche di interconnessione fra computer furono storicamente messe a punto dai costruttori di sistemi informativi, primo *fra* tutti IBM. Le soluzioni che ne risultavano erano però delle reti di computer "chiuse", ossia costituite da apparati tutti dello stesso costruttore e incapaci di comunicare con macchine di altra origine.

L'avvento di sistemi informativi "aperti" (tali cioè da poter realizzare comunicazioni reciproche in qualunque combinazione) era da una parte facili tate dall'emergere di ARPAnet e delle reti pubbliche per dati, dall'altra si scontrava con problemi che andavano oltre il semplice trasferimento di bit da un punto a un altro. Si pensi ad esempio ai problemi connessi con l'uso di diversi alfabeti per rappresentare testi; diverse strutture dei *file System;* diverse procedure di stampa, di *login* e tanti altri ancora.

Il problema fu affrontato in modo sistematico dall'ISO che, in collaborazione col CCITT (oggi ITU -T), diede avvio alla fine degli anni 70 al prog etto *Open System Interconnection* (OSI). Il Modello di Riferimento definito da OSI può essere considerata un'architettura che non dice tanto come il sistema è costituito al suo interno, ma come si presenta verso il mondo esterno, ossia verso gli altri sistemi.

Emanato dall'**ISO (International Organization for Standardization)** nella sua versione definitiva nel 1984, descrive un modello di architettura versatile ed efficiente basato su una serie di **l**ivelli o strati gerarchici in cui adattare gli standard esistenti o indirizzare quelli futuri.

Il modello OSI è, quindi, un formalismo astratto che non entra nel dettaglio della definizione di servizi o protocolli specifici di una architettura. Esso definisce una struttura in cui le funzione logiche necessarie alla comunicazione tra sistemi sono suddivise in sette livelli funzionali ordinati gerarchicamente.

I principi dei progetto seguiti durante lo sviluppo del modello furono i seguenti:

1. Ogni livello debe avere una funzione ben definita, in modo da minimizzare il numero dei livelli e le funzioni svolte da ciascuno.

2. Il passaggio delle informazioni avviene solo tra livelli adiacenti, attraverso opportune interfacce; ogni livello fornisce servizi al livello superiore utilizzando quelli resi disponibili dal livello superiore.

3. La comunicazione tra sistemi diversi avviene tra livelli paritari, attverso un protocollo relativo a ciascun livello.

1.1.1 Elenco e funzioni dei livelli OSI.

I livelli sono sette, e ognuno offre servizi più evoluti nella misura in cui si sale nella gerarchia.

- ***Livello Fisico :*** Riguarda la trasmissione dei bit sul canale fisico di trasmissione.

  Convolge aspetti di tipo:

  -Elettrico (linee comunicazione, propagazione onde)

  -Communicacione (simplex, half-, full-duplex, …)

  -Mecanico (standards connettori, …)

- ***Livello Collegamento :***
    - Trasforma la linea fisica in una linea in cui gli errori di trasmissione vengano sempre segnalati.
    - Divide le informazioni in pacchetti e li trasmette attraverso il mezzo fisico, attendendo un segnale di "avvenuta ricezione" (ack).
    - Gestisce l'eventuale duplicazione dei frame ricevuti, causata dalla perdita dell'ack.
    - Sincronizza un mittente veloce con un ricevente lento.
    - Gestisce l'accesso al canale di trasmissione condiviso.

- ***Livello Rete :***
    - Controlla il cammino e il flusso di pacchetti.
    - Gestisce la congestione della rete.
    - Gestisce l'accounting dei pacchetti sulle reti a pagamento.
    - Implementa l'interfaccia necessaria alla comunicazione di reti di tipo diverso.

- ***Livello Transporto:***
    - Accetta dati dal livello superiore, li spezza in parti più piccole e le trasmette, assicurando un servizio privo di errori e l'ordine corretto di ricomposizione.
    - Gestisce la diffusione di messaggi a più destinazioni.
    - Fornisce il servizio di recapito dei messaggi senza garanzia sull'ordine del loro arrivo al destinatario.

- ***Livello Sessione:***

    - Controlla il dialogo tra due macchine: la comunicazione non può essere sempre full-duplex, questo layer tiene traccia di chi è il turno attuale.
    - Gestisce il controllo dei token.
    - Gestisce la sincronizzazione del trasferimento dei dati.

- ***Livello Presentazione:***

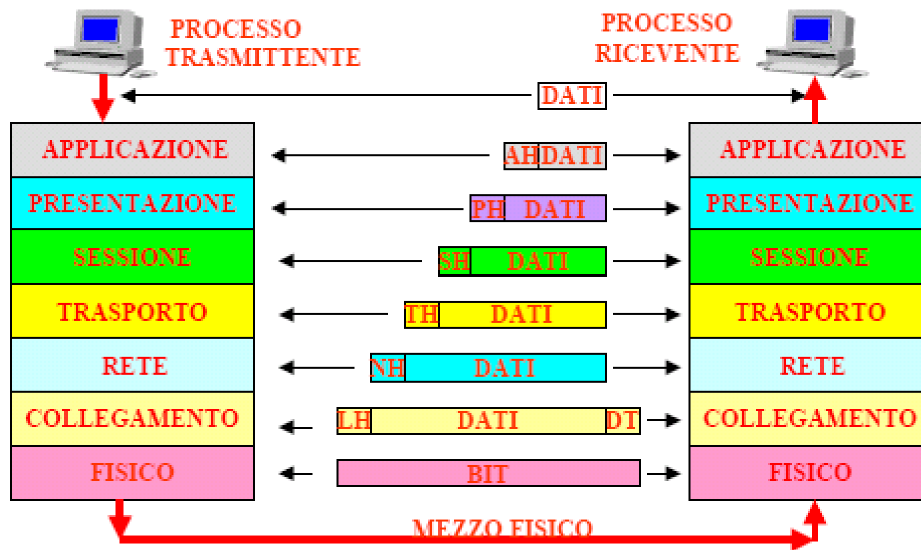
    - La funzionalità di queste livello si limitano alla traduzione dei dati che viaggiano sulla rete in formati astratti. Queste informazioni vengono poi riconvertite nel formato proprietario della macchina destinataria.

- ***Livello Applicazione:***

    - I servizi di questo livello sono completamente legati alle applicazioni:

        - Quali dati trasmettere
        - Quando trasmettere
        - Dove trasmettere / a chi
        - Significato di bits/bytes.

Esempi di applicazioni sono: File Transfer, Posta elettronica, World Wide Web, Multimedialità, File System distribuiti.



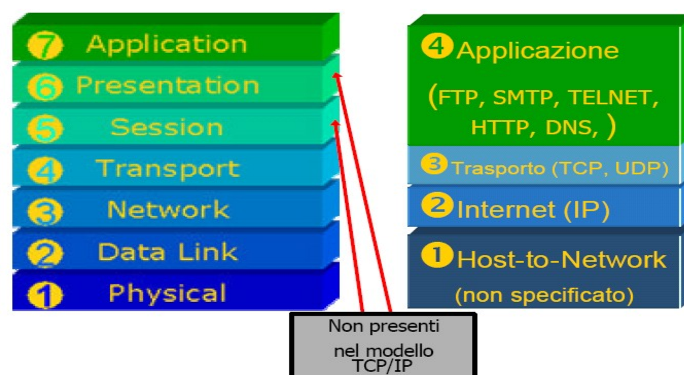
1.2 L`Arquittetura TCP/IP

I protocolli di rete sono normalmente sviluppati e organizzati su più livelli, il caso di TCP/IP è diverso, perchè sebbene si sviluppa secondo la struttura ISO/OSI ,comprende solo quattro livelli e ciascun livello è responsabile di un aspetto della comunicazione TCP/IP.

Applications	Telnet, FTP, e-mail, etc.
Transport	TCP, UDP
Network	IP, ICMP, IGMP
Link	Device drivers e schede di interfaccia

1.2.1 Comparazione tra modello OSI e modello TCP/IP.

Il motivo dalla fine avere fatto l`elenco del modello TCP/IP, ha di tre ragioni fondamentali. Il primo è chè quando nacque il modello OSI, TCP/IP era gia presente nel mondo accademico. Insoma, lo stack TCP/IP è enormemente più semplice dello stack OSI. E per ultimo, il motivo più determinante è chè mentre OSI parte dei livelli, TCP/IP parte dei Protocolli.



1.2.2 Il Protocolo di rete IP.

Il protocollo IP, descritto nella RFC 791, nasce negli anni '70 grazie a una serie di ricerche fatte dalle università americane su richiesta del ministro della difesa, allo scopo di realizzare una rete in grado di trasportare diversi tipi di informazioni.

IP offre un servizio di tipo connection-less , cioè una applicazione trasmette i pacchetti IP (detti anche “datagram”) senza stabilire una connessione preliminare fisica o logica tra mittente e destinatario (al contrario di ciò che avviene nella telefonia). Questo implica che per esempio il cammino (path) di ciascun pacchetto è instradato in modo indipendente dagli altri. I pacchetti possono anche seguire cammini diversi e indipendenti.

Il protocollo è *unreliable* , ovvero IP non è in grado di riconoscere la perdita di un pacchetto. Tuttavia in caso di congestione e di perdita, è il protocollo di livello superiore (livello Transport) , il quale si incarica di effettuare la ritrasmissione delle porzioni di dati perse.

L'indirizzamento IP permette di identificare ogni host all'interno di una rete TCP/IP. Grazie all'utilizzo delle classi di indirizzi ed al subnetting è possibile organizzare e gestire in modo più efficiente il proprio network.

Un indirizzo IP, chiamato anche indirizzo logico, rappresenta un identificativo software per le interfacce di rete, esso viene utilizzato in combinazione con l'indirizzo fisico (MAC), il quale consente di determinare in modo univoco ogni interfaccia di un dispositivo di rete. Un IP Address è un numero di 32 bit suddiviso in quattro gruppi da 8 bit ciascuno, la forma con la quale viene solitamente rappresentato è detta decimale puntata (Dotted Decimal).

Essendo ogni numero rappresentato da 8 bit, può assumere un range di valori da 0 a 255. Utilizzando 32 bit per indirizzo è possibile avere 4.294.967.296 combinazioni di indirizzi differenti. In realtà esistono alcuni indirizzi particolari, di conseguenza non tutti i valori sono disponibili al fine di identificare un host nella rete.

Classi di indirizzi:

Per permettere una migliore organizzazione della rete, gli indirizzi disponibili sono stati suddivisi in classi in base alle dimensioni del network da gestire. In questo modo si verrando utilizzate le classi più adatte ad alla dimensioni della rete, con conseguente minore spreco di ip address. Sono disponibili cinque classi di indirizzi IP, di cui solo le prime tre possono essere utilizzate per assegnare indirizzi agli host.

Indirizzi di classe A
Il valore del primo ottetto è compreso tra 1 e 126 (I primi otto bit di questo indirizzo saranno: 0*****).
E' rappresentata da indirizzi di tipo: Rete.Host.Host.Host ovvero 8 bit per la identificare la rete (di cui il primo fisso) e 24 per identificare gli host. Permette di ottenere 126 reti formate da 16.774.214 host ciascuna.

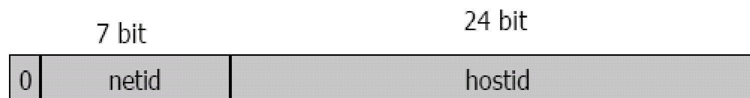
Indirizzi di classe B
Il valore del primo ottetto è compreso tra 128 e 191 (I primi otto bit di questo indirizzo saranno: 10*****).
E' rappresentata da indirizzi di tipo: Rete.Rete.Host.Host ovvero 16 bit per la identificare la rete(di cui i primi due fissi) e 16 per identificare gli host. E' possibile ottenere 16.384 reti formate da 65.534 host ciascuna.

Indirizzi di classe C
Il valore del primo ottetto è compreso tra 192 e 223 (I primi otto bit di questo indirizzo saranno: 110*****).
E' rappresentata da indirizzi di tipo: Rete.Rete.Rete.Host ovvero 24 bit per la identificare la rete (di cui i primi tre fissi) e 8 per identificare gli host. E' possibile ottenere 2.097.152 reti con 254 host ciascuna.

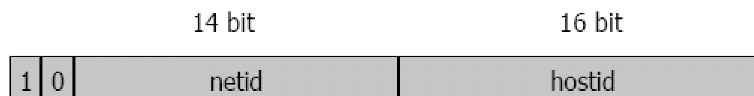
Indirizzi di classe D
Il valore del primo ottetto è compreso tra 224 e 239 (I primi otto bit di questo indirizzo saranno: 1110****).
Sono indirizzi di rete riservati ai gruppi multicast e non assegnabili ai singoli host.

Indirizzi di classe E
Il valore del primo ottetto è compreso tra 240 e 255 (I primi otto bit di questo indirizzo saranno: 1111****).
Sono indirizzi riservati per usi futuri.

Classe A **(0.0.0.0 - 127.255.255.255**



Classe B **(128.0.0.0 - 191.255.255.255)**



Classe C **(192.0.0.0 - 223.255.255.255)**

Classe D 28 bit **(224.0.0.0 - 239.255.255.255)**

Classe E **(240.0.0.0 - 255.255.255.254)**

1.2.3 Caratteristiche protocollo IPv6.

IPv6 è stato ideato come evoluzione e non come rivoluzione di IPv4. I cambiamenti principali introdotti nel nuovo protocollo si possono raggruppare nelle seguenti categorie:

- **Capacità di instradamento e indirizzamento espanso.**
  IPv6 aumenta la dimensione dell'indirizzo IP da 32 a 128 bit per supportare più livelli gerarchici di indirizzamento ed un numero molto più grande di nodi indirizzabili.

- **Una semplificazione del formato dell'*header*.**
  Dal confronto delle intestazioni della versione 4 e della versione 6 di IP si può verificare che alcuni campi dell'*header* IPv4 sono stati rimossi o resi opzionali, per ridurre il peso dell'elaborazione del datagramma e per contenere il più possibile l'occupazione di banda dovuta all'intestazione, nonostante la maggiore dimensione degli indirizzi. Anche se gli indirizzi IPv6 sono quattro volte più lunghi di quelli IPv4, l'*header* IPv6 è solo due volte più grande di quello Ipv4.

- **Supporto migliorato per le opzioni.**
  Cambiamenti nel modo in cui le opzioni dell'*header* IP sono codificate permettono un *forwarding* più efficiente, limiti meno stringenti sulla lunghezza delle opzioni e maggiore flessibilità nell'introduzione di nuove opzioni. Il loro uso diventa realmente possibile.

- **Meccanismo di individuazione dei flussi.**
  E´ stata aggiunta una nuova funzionalità per permettere l'individuazione dei pacchetti appartenenti a particolari flussi di dati per i quali il mittente richiede un trattamento speciale.

- **Possibilità di estensioni future per il protocollo.**
  Forse il cambiamento più significativo in IPv6 è l'abbandono di un protocollo che specifica completamente tutti i dettagli a favore di un protocollo che consente di inserire delle estensioni.

- **Ottimizzazione delle funzioni di controllo.**
  Il protocollo ICMP (*Internet Control Message Protocol*) per IPv6 comprende al suo interno la gestione dei Gruppi Multicast ed i cosiddetti meccanismi di *Neighbor Discovery*: tra questi sono particolarmente importanti i meccanismi di auotoconfigurazione dei terminali e quelli di risoluzione indirizzi.

- **Un nuovo tipo di indirizzo chiamato indirizzo *anycast*.**
  Questo indirizzo identifica un insieme di nodi, ma un datagramma spedito a tale indirizzo viene inoltrato a uno solo di essi.
- Capacità di autenticazione e *privacy*.
- Allineamento su 64 bit anziché su 32 bit.

1.2.3.1 Formato dell`Header.

Il nuovo *header* di Ipv6 [RFC2460] è illustrato in figura 1, mentre in figura 2 è stato riportato quello IPv4 alla fine di facilitare il confronto tra i due protocolli.

- *version (4 bit)*: indica la versione del protocollo, quindi conterrà il numero 6.
- *DS byte (8 bit)*: questo campo è utilizzato dalla sorgente e dai router per identificare i pacchetti che appartengono a una stessa classe di traffico e quindi distinguere tra loro i pacchetti con diversa priorità.
- *flow label (20 bit)*: etichetta un flusso di dati.
- *payload length (16 bit)*: indica la lunghezza del campo dati del pacchetto.
- *next header (8 bit)*: identifica il tipo di header che segue immediatamente quello IPv6.
- *hop limit (8 bit)*: è decrementato di uno da ogni nodo che inoltra il pacchetto. Quando il suo valore è nullo il pacchetto viene scaricato.
- *source address (128 bit)*: indirizzo della sorgente.
- *destination address (128 bit)* : indirizzo di destinazione.

Rispetto IPv4, il formato dell'intestazione è più semplice e questo permette migliori prestazioni.

La scelta di eliminare il checksum deriva dal fatto che esso è già calcolato a livello 2 e, dato il tasso di errore delle reti attuali, tale controllo è sufficiente. Si ottengono così migliori prestazioni poi ché i router non devono più ricalcolarlo per ogni pacchetto.

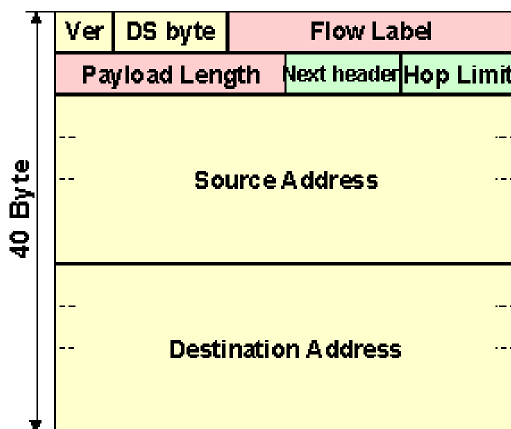


Figura 1

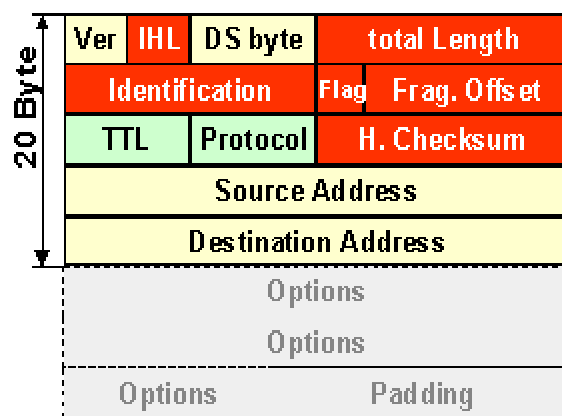


Figura 2

Il campo hop limit indica il numero massimo di nodi (*hop*) che un pacchetto può attraversare prima di giungere a destinazione. In IPv4 tale campo è espresso in secondi (TTL: Time To Live), anche se ha la stessa funzione. Il cambiamento è stato fatto per due motivi. Innanzi tutto per la semplicità di realizzazione: infatti, anche in IPv4 i *router* traducono i secondi in numero di *hop* per poi ritradurli in secondi. In secondo luogo ci si svincola dalle caratteristiche fisiche (quali la banda) della rete. Poiché il campo hop limit è di 8 bit, il numero massimo di nodi attraversabili da un pacchetto è 255.

L'hop limit può servire anche ad un altro scopo: cercare tra un gruppo di *server* che svolgono la stessa funzione quello più vicino.

1.2.3.2 Evoluzione di ICMP (ICMPv6).

La nuova versione di ICMP, chiamata ICMPv6 RFC[2463] ingloba il vecchio protocollo (tranne le opzioni meno usate) e il protocollo IGMPv2 che IPv4 usa per la gestione del *multicast*. I tipi di messaggi definiti per ora sono 14 e si possono raggruppare in base alle funzioni che svolgono. Tutti i messaggi hanno lo stesso formato generale.

- type indica il tipo del messaggio;
- code è usato solo in alcuni casi;
- checksum copre tutto il pacchetto ICMP e i campi fissi dell'intestazione IPv6;
- message body è un campo di lunghezza variabile che contiene il corpo del messaggio.

ICMPv6 include le procedure per la gestione dei messaggi di errore, per il *ping* e per la *MTU discovery:*

Messaggi di errore : Quando un nodo IPv6 scarta un pacchetto, invia un messaggio di errore alla sorgente. I tipi di messaggio sono quattro:

Procedura di Ping : Ping indica una semplice procedura che consente di verificare la raggiungibilità di una destinazione disponibile in IPv6 come in IPv4.

Procedura di MTU Discovery :La procedura di *MTU Discovery* permette alle stazioni di conoscere la dimensione massima del pacchetto che può essere utilizzata nella comunicazione verso una destinazione scelta. A tale scopo la sorgente invia un pacchetto con MTU pari a quella del primo link, se tale pacchetto supera la misura della MTU dell'*hop* successivo viene scartato e viene mandato alla sorgente un messaggio ICMP di errore (`Type`=2, *message too big*).

1.2.3.3 L`indirizzo IPv6

L'indirizzo IPv6 è costituito da 128 bit (16 byte), viene descritto da 8 gruppi di 4 numeri esadecimali che rappresentano 2 byte ciascuno (quindi ogni numero varia tra 0 e 65535) separati dal simbolo "due punti". Un esempio di indirizzo IPv6 è 2001:0DB8:0000:0000:0000:0000:0000:0001, che può essere abbreviato in 2001:DB8::1 (i due punti doppi stanno a sostituire la parte dell'indirizzo che è composta di soli zeri consecutivi. Si può usare una sola volta, per cui se un indirizzo ha due parti composte di zeri la più breve andrà scritta per esteso).

I dispositivi connessi ad una rete IPv6 ottengono un indirizzi di tipo unicast globale vale a dire che i primi 48 bit del suo indirizzo sono assegnati alla rete a cui esso si connette, i successivi 16 bit identificano le varie sottoreti a cui l'host è connesso. Gli ultimi 64 bit sono ottenuti dall'indirizzo MAC dell'interfaccia fisica.

Gli indirizzi Ipv4 sono facilmente trasformabili in formato IPv6. Ad esempio, se l'indirizzo decimale IPv4 è 135.75.43.52 (in esadecimale, 0x874B2B34), può essere convertito in: 0000:0000:0000:0000:0000:0000:874B:2B34 o più brevemente ::874B:2B34. Anche in questo caso è possibile l'uso della notazione ibrida (Ipv4 compatible-address) usando la forma ::135.75.43.52.

Indirizzi speciali

È stato definito un certo numero di indirizzi con significati particolari. La tabella seguente ne elenca alcuni nella forma CIDR notation.

- ::/128 – L'indirizzo composto da tutti zeri viene utilizzato per indicare *qualsiasi indirizzo* e viene utilizzato esclusivamente a livello software.
- ::1/128 – l'indirizzo di loopback è un indirizzo associato al dispositivo di rete che ripete come eco tutti i pacchetti che gli sono indirizzati. corrisponde al 127.0.0.1 dell' IPv4.
- ::/96 – è utilizzato per interconnettere le due tecnologie IPv4/IPv6 nelle reti ibride.
- ::ffff:0:0/96 – L'indirizzo Ipv5 mapped address è utilizzato nei dispositivi dual-stack hosts.
- fe80::/10 – Il prefisso link-local specifica che l'indirizzo è valido esclusivamente sullo specifico link fisico.
- fec0::/10 – Il prefisso site-local specifica che l'indirizzo è valido esclusivamente all'interno dell'organizzazione locale. Il suo uso è stato sconsigliato nel Settembre del 2004 con il RFC3879 e i sistemi futuri non ne dovrebbero implementare il supporto.
- ff00::/8 – Il prefisso di multicast è utilizzato per gli indirizzi di multicast.

Capitolo II. I Reti Wireless.

Sin dagli anni '70, surge la possibilità di collegare computer fra loro senza usare cavi. Surge anche la possibilità di accedere ad internet, con il proprio portatile, senza dipendere da cavi di collegamento.

Per essempio, un scenario tipico è quello di un ufficio che abbia installato uno o più punti di acceso wireless, è sufficiente entrare nel raggio d`azione di uno di questi AP, per potere accedere a internet, e cosi spedire un email, senza dipendere di un cavo.

In queste senso, la tecnologia wireless è orientata ad offrire connettività ad una LAN cablata preesistente per soddisfare requisiti quali la mobilità, la rilocazione, la connessione ad-hoc e la copertura di aree difficili da cablare.

Sustituire completamente l`uso di cavi di collegamento, in altri ambiti, poè essere interessante, gia chè il cablaggio trazionale poè risultare troppo costoso o scomodo.

Per tanto i sistemi wireless sono comodi e spesso meno costosi da sviluppare rispetto ai sistemi fissi. Ma non sono perferfetti. Esistono dei limiti e difficoltà tecniche chè possono impedire lo sfruttamento delle piene potenzialità di tali tecnologie.

Le wireless LAN sono state standardizzate nel giugno 1997 dal Comitato IEEE. Lo standard include requisiti dettagliati per il livelli fisico e per la parte inferiore del livello data link, ovvero il MAC (Medium Access Control), secondo la terminologia introdotta dallo standard IEEE 802.

2.1 Reti WLAN.

WLAN è l'acronimo di (*Wireless Local Area Network*) un sistema di comunicazione dati via etere, usato in aggiunta o sostituzione di una rete LAN cablata.

Una rete WLAN opera usando onde radio su frequenze non soggette a licenza, con una copertura localizzata e solitamente ristretta ad un centinaio di metri.

Le reti WLAN si basano tipicamente sugli standard IEEE 802.11, 802.11b e 802.11g, con prestazioni comprese tra 1 Mbps ed i 54 Mbps.

Esistono diverse configurazioni di rete WLAN, la più semplice è quello di tipo "Peer to Peer" dove le schede di rete wireless di due computer comunicano tra di loro. Altri tipi di configurazione possibili sono:

- *Punto di accesso e client*. Il punto di accesso o Access Point (AP) è composto da un dispositivo hardware e software che consente il collegamento ad una rete fissa, ad uno o piu dispositivi client, solitamente costituiti da PC dotati di scheda di rete wireless. Questa configurazione serve connessioni sino a 150 metri in locali chiusi e sino a 300 metri in spazi aperti.

- *Roaming o Punti Accesso Multipli.* Quando lo spazio da coprire è elevato, è possibile aggiungere nuovi Access Point per estendere l'area coperta dalla rete WLAN. I client che si muovono nell'area coperta dalla rete senza fili possono passare da un punto di accesso all'altro grazie ad un meccanismo automatico di roaming.

- *Punti di estensione.* I punti di estensione o Extension Point (EP) amplificano il segnale di un punto di accesso estendendone il raggio di copertura.

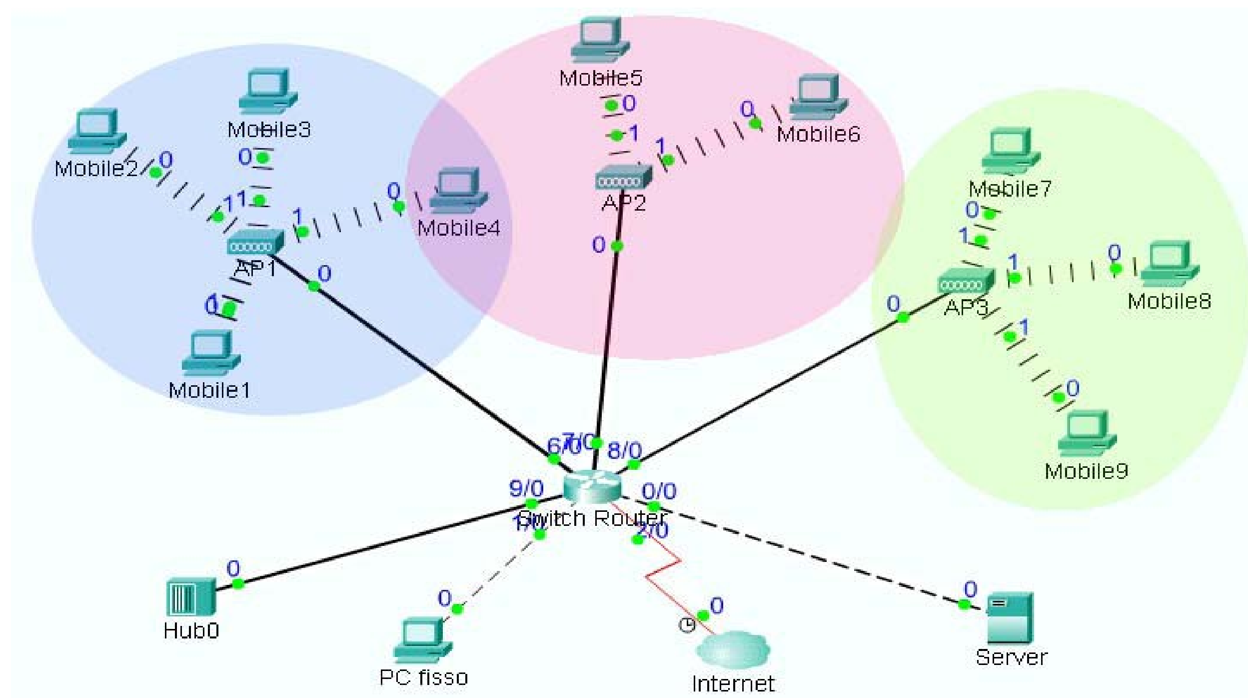
Antenna direzionale. Se i punti di estensione non possono essere usati, allora è possibile utilizzare le antenne direzionali. Con le antenne direzionali è possibile inviare il segnale a distanze superiori al chilometro.

2.1.1 - Estensioni di Reti Locali.

In generale, una WLAN verrà collegata a una LAN situata negli stessi ambienti. Si parla quindi in questo caso di un'estensione alle rete locale esistente.

Esistono diversi mode di possibili di organizzazione, per essempio è molto comune trovare una configurazione dove insieme sono più celle.

Per essempio:



In questo caso vi saranno più Access Point interconnessi da una rete locale cablata. Ogni AP supporta un certo numero di sistemi terminali wireless nella rispettiva area di trasmissione.

Tale configurazione prende il nome di *Infrastructure Basic Service Set* ***(BSS)***. L'AP può essere definito come una speciale stazione ripetitore che fornisce sincronizzazione e coordinazione, si occupa dell'instradamento dei pacchetti trasmessi e permette di connettere la rete wireless alla rete cablata. Gli AP possono essere implementati sia in hardware che in software.

Lo standard 802.11 (di cui parleremo più avanti) si riferisce alla configurazione con un singolo AP con il nome di BSS. L'uso di più AP è chiamato ***ESS*** (*Extended Service Set*) e definisce 2 o più BSS connesse alla stessa rete fissa (a ciascun AP viene dato un differente canale).

Quando un messaggio viene spedito da un dispositivo wireless in una BSS verso una differente BSS attraverso la rete fissa, si dice che viene spedito attraverso il ***DS*** (*Distribution System*) di solito implementato usando la tradizionale rete Ethernet cablata.
Gli AP sono dotati di una porta Ethernet, e sono quindi a tutti gli effetti dei bridge tra WLAN e Ethernet.

2.1.2 - Sistemi di trasmissione di Reti Locali a Onde Radio

Generalmente le wireless LAN sono suddivise in categorie a seconda della tecnica di trasmissione utilizzata. La maggior parte delle WLAN sono della seguente topollogia, A onde Radio.

Con il termine radio frequenze, spesso abbreviato con l'acronimo RF, ci si riferisce a quella singola porzione dello spettro che include le onde elettromagnetiche le cui frequenze sono comprese tra i 3 KHz e i 300 GHz, lo spettro cioè delle onde radio.

Questa tecnica è la più diffusa nella realizzazione di reti WLAN e prevede una configurazione a più celle adiacenti (vedi fig.1.2) che utilizzano frequenze centrali differenti nell'ambito della stessa banda in modo da evitare interferenze. In una determinata cella, la topologia può essere a punto di accesso (AP) o peer-to-peer; l'Access Point è connesso a una rete locale cablata così che una stazione mobile è in grado di comunicare anche con un host della rete fissa.

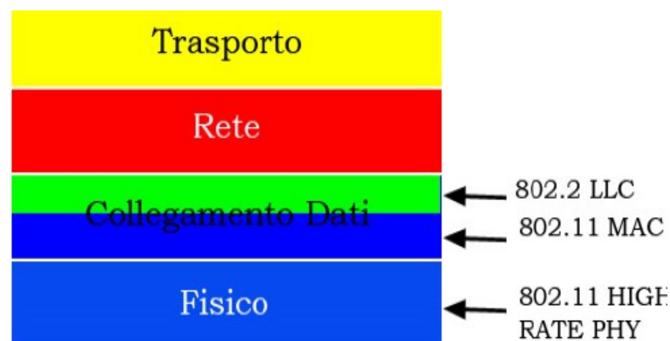
Una delle caratteristiche principali dell'AP è la possibilità di effettuare il *roaming* automatico dei terminali wireless, ovvero permettere il passaggio da una cella ad un'altra senza la perdita della connessione.
Invece in una topologia di tipo peer-to-peer non esiste alcun AP ma viene utilizzato un algoritmo MAC, come il CSMA, per controllare gli accessi. Essa è appropriata per le reti ad hoc.

In Europa le frequenze portanti riservate per questo tipo di reti sono quella a 2,4 Ghz e quella a 5 Ghz. Quest'ultima ha una maggiore larghezza di banda e quindi maggiore capacità ma ciò si ripercuote sul costo dei dispositivi.
Le velocità di trasmissione vanno da 1 Mb/s fino agli attuali 54 Mb/s mentre è in fase di studio una tecnologia che può raggiungere i 100 Mb/s.

2.2 Lo Standard 802.11

L'IEEE (Institute of Electrical and Electronic Engineers) è l'ente degli Stati Uniti nato nel 1963, con sede a New York che definisce i modelli di rete e metodi di accesso, e sviluppa ed approva gli standard di un'ampia gamma di tecnologie.

Gli standard di networking di IEEE portano il numero 802, mentre quelli wireless ne sono un sottoinsieme, con il numero 11. Il primo standard wireless di IEEE, adottato nel 1997, era semplicemente chiamato IEEE 802.11; era uno standard di trasmissione radio a 2,4 Ghz con un throughput2 massimo di 2 Mbps. Una revisione dello standard a velocità 11 Mbps, fu chiamata 802.11 High Rate. Nel 1999 lo standard 802.11 High Rate venne chiamato 802.11b. In seguito sono stati aggiunti come standard sia 802.11a, sia 802.11g.



802.11 sopra Modello ISO/OSI

2.2.1 Bande ISM

La Bande ISM è una bande chè molti governi hanno mantenuto libera, con il scopo di essere usata solo per i sectori Industriale, Scientifico e Medico. Esse possono essere usate liberamente da chiunque, senza dover richiedere licenze, a patto di rispettare precisi limiti di potenza e limitare anche la interferenze fra i diversi dispositivi.

Sono diversi i bandi disponibili per essempio in USA o in Europa. In USA sono disponibile 3 bande, una a 902 MHz, altra a 2.4 GHz e l`ultima a 5725 Ghz.

In Europa solo è disponibile la bande di 2.4 GHz.

Inoltre, all`aumentare della frequenza, aumentano gli effetti di reflessione ed assorbimento delle onde elettromagnetiche, e di conseguenza diminuiscono le distanze raggiungibili. In particolare, a 2.4 GHz è possibile coprire una distanza 4 volte superiore che a 5 GHz.

Per questi ed altri motivi, la maggioranza degli standard utilizza la banda ISM a 2.4 GHz.

2.2.2 Lo strato fisico

Lo standard 802.11 del 1997, specificava tre tecniche di trasmissione, tutte ad 1 o 2 Mbps. A detta di molti, le prestazioni fornite erano insufficienti. Il comitato si mise di nuovo al lavoro, e nel 1999 furono approvati 2 nuovi standard:

· **802.11b**, compatibile con 802.11, aggiungeva a quest'ultimo due nuove velocità: 5.5 Mbps e 11 Mbps.

· **802.11a**, che, sfruttando una delle più versatili tecniche di modulazione (QAM-64), poteva raggiungere i 54 Mbps. 802.11a usava però la banda a 5 GHz, e quindi, 2 anni dopo, il comitato propose una sua variante, **802.11g**, consentendo di raggiungere i 54

Mbps nella banda ISM tradizionale a 2.4GHz, e mantenendo la compatibilità verso il basso con i dispositivi 802.11b (l'approvazione di quest'ultimo standard è prevista per Giugno 2003).

La seguente tabella riassume le caratteristiche fisiche delle varianti 802.11 proposte:

Protocollo	Modulazione	Velocità	Banda utilizzata
802.11	Infrarosso diffuso	1 o 2 Mbps	-
	FHSS	1 o 2 Mbps	2.4 GHz ISM
	DSSS	1 o 2 Mbps	2.4 GHz ISM
802.11a	OFDM	54 Mbps (max)	5.2 GHz UNII
802.11b	HR-DSSS	11 Mbps (max)	2.4 GHz ISM
802.11g	OFDM	54 Mbps (max)	2.4 GHz ISM

La velocità viene adattata dinamicamente sulla base del rapporto segnale/disturbo.
In questa sede ci si limita a notare che, al fine di evitare interferenze fra dispositivi radio che sfruttano la stessa banda, si ricorre a tecniche di spread spectrum, che consistono nel distribuire il segnale su una banda molto più larga del necessario, in modo che esso appaia come rumore ai dispositivi non interessati. Ad es. FHSS (Frequency Hopping Spread Spectrum) ottiene lo scopo saltando ad intervalli regolari da una frequenza portante all'altra in modo pseudocasuale; solo i dispositivi sintonizzati sullo stesso seme e intervallo (dwell time) rilevano un segnale netto. Ciò incrementa anche la sicurezza e la reiezione al multipath fading.

2.2.3 Protocollo MAC

Lo standard 802.11 definisce il formato delle trame scambiate. Ciascuna trama è costituita da un'intestazione (detta *MAC header*, lunga 30 bte), da un corpo e da un *FCS* (*Frame Sequence Check*) che permette la correzione di errore.

FC (2)	D/ID (2)	Indirizzo 1 (4 byte)	Indirizzo 2 (4 byte)	Indirizzo 3 (4 byte)	SC (2)	Indirizzo 4 (4 byte)	Corpo della trama (da 0 a 2312 byte)	FCS (2)

Descrizione di questi campi:

- **FC** (*Frame Control*): questo campo di due byte è costituito dalle seguenti informazioni:

Versione del protocollo (2 bits)	Tipo (2 bits)	Sotto tipo (4 bits)					
To DS (1 bit)	From DS (1 bit)	More Frag (1 bit)	Retry (1 bit)	Power Mgt (1 bit)	More Data (1 bit)	WEP (1 bit)	Order (1 bit)

  - **Versione del protocollo** : questo campo di 2 bit permetterà di considerare le evoluzioni della versione dello standard 802.11. Il valore è pari a zero per la prima versione.

  - **Tipo** e **Sotto tipo** : questi campi, rispettivamente di 2 e 4 bit, definiscono il tipo e il sotto tipo delle trame . Il tipo *gestione* corrisponde alle richieste di associazione nonché ai messaggi di annuncio dei punti di accesso. Il tipo *controllo* è usato per l'accesso al media per richiedere le autorizzazioni per emettere. Infine il tipo *dati* riguarda gli invii di dati (la maggior parte del traffico).

  - **To DS** : questo bit vale 1 quando la trama è destinata al sistema di distribuzione (*DS*), vale zero negli altri casi. Ogni trama inviata da una stazione a destinazione di un punto d'accesso ha quindi un campo *To DS* posizionato a 1.
  - **From DS** : questo bit vale 1 quando la trama proviene dal sistema di distribuzione (*DS*), vale zero negli casi. Quindi, quando i due campi *To* e *From* sono posizionati a zero si tratta di una comunicazione diretta tra due stazioni (modalità *ad hoc*).
  - **More Fragments** (*frammenti supplementari*): permette di indicare (quando vale 1) che restano dei frammenti da trasmettere
  - **Retry** : questo bit specifica che il frammento in corso è una ritrasmissione di un frammento precedentemente inviato (e sicuramente perso)
  - **Power Management** (*gestione d'energia*): indica, quando è a 1, che la stazione che ha inviato questo frammento entra in modalità di gestione dell'energia
  - **More Data** (*gestione d'energia*): questo bit, usato per la modalità di gestione dell'energia, è usato dal punto di accesso per specificare ad una stazione che delle trame supplementari sono stoccate in attesa.
  - **WEP** : questo bit indica che l'algoritmo di codifica WEP è stato usato per cifrare il corpo della trama.
  - **Order** (*ordine*): indica che la trama è stata inviata usando la classe di servizio strettamente ordinata (*Strictly-Ordered service class*)
- **Durata / ID** : Questo campo indica la durata d'utilizzo del canale di trasmissione.
- **Campi indirizzi** : una trama può contenere fino a 3 indirizzi oltre all'indirizzo di 48 bit.
- **Controllo di sequenza** : questo campo permette di distinguere i diversi frammenti di una stessa trama. E' composto da due sotto campi che permettono di riordinare i frammenti:
  - Il *numero di frammento*
  - Il *numero di sequenza*
- **CRC** : una somma di controllo che serve a verificare l'integrità della trama.

La tabella sottostante riassume i tipi e i sotto tipi di trame incapsulati nel campo di controllo della trama dell'intestazione MACnbsp;:

Tipo	Descrizione del tipo	Sotto tipo	Descrizione del sotto tipo
00	Management (gestione)	0000	Association request (richiesta di associazione)
00	Management (gestione)	0001	Association response (risposta d'associazione)
00	Management (gestione)	0010	Reassociation request (richiesta ri-associazione)
00	Management (gestione)	0011	Reassociation response (risposta di ri-associazione)
00	Management (gestione)	0100	Probe request (richiesta di inchiesta)
00	Management (gestione)	0101	Probe response (risposta di inchiesta)
00	Management (gestione)	0110-0111	Reserved (riservato)
00	Management (gestione)	1000	Beacon (tag)
00	Management (gestione)	1001	Annoucement traffic indication message (*ATIM*)
00	Management (gestione)	1010	Disassociation (dissociazione)
00	Management (gestione)	1011	Authentication (autentificazione)
00	Management (gestione)	1100	Deauthentication (disautentificazione)
00	Management (gestione)	1101-1111	Reserved (riservato)
01	Control (controllo)	0000-1001	Reserved (riservato)
01	Control (controllo)	1010	Power Save (PS)-Poll (economia d'energia)
01	Control (controllo)	1011	Request To Send (RTS)
01	Control (controllo)	1100	Clear To Send (CTS)
01	Control (controllo)	1101	ACK
01	Control (controllo)	1110	Contention Free (CF)-end
01	Control (controllo)	1111	CF-end + CF-ACK
10	Data (dati)	0000	Data (dati)
10	Data (dati)	0001	Data (dati) + CF-Ack
10	Data (dati)	0010	Data (dati) + CF-Poll
10	Data (dati)	0011	Data (dati) + CF-Ack+CF-Poll
10	Data (dati)	0100	Null function (no Data (dati))
10	Data (dati)	0101	CF-Ack
10	Data (dati)	0110	CF-Poll
10	Data (dati)	0111	CF-Ack + CF-Poll
10	Data (dati)	1000-1111	Reserved (riservato)
11	Data (dati)	0000-1111	Reserved (riservato)

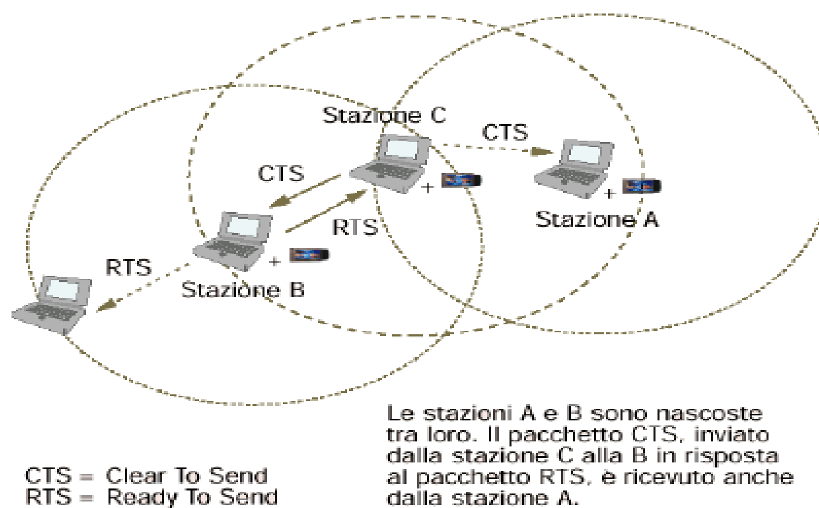
2.2.4 Metodo di acceso al medio RTS/CTS.

RTS/CTS (Request to Send/Clear to Send) è un meccanismo usatto per il protocollo di rete wireless 802.11 per reduzire i colissioni dei “frames” introdutti per il problema del terminale nascosto.

“Ma, ¿cual è il problema del Terminale nascosto?”

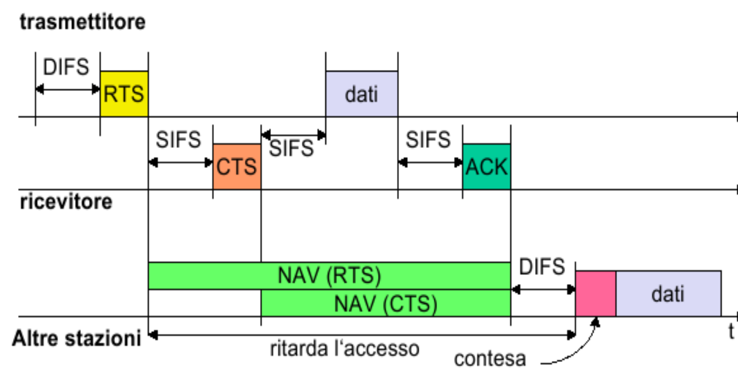
La copertura radio delle stazioni A e B è tale che queste non sono in visibilità reciproca, e quindi non possono accorgersi se l'altra sta già trasmettendo, rendendo inapplicabile il meccanismo di Collision Avoidance basato sull'attesa di un periodo di silenzio pari al DIFS.

Nel caso di stazioni in visibilità, la finestra temporale in cui si può verificare collisione è pari al tempo di propagazione tra le stazioni, mentre nel caso di assenza di visibilità, si estende alla durata della trasmissione di una intera trama, aumentando considerevolmente la probabilità di una collisione.



Per evitare questo fenomeno, le stazioni possono configurare un parametro, che sancisce l'intervento di un diverso meccanismo di prenotazione delle risorse, qualora la dimensione della trama superi un certo valore.

In tal caso la trasmissione è preceduta da uno scambio RTS-CTS (Request To Send - Clear To Send) in cui chi deve trasmettere (B nell'esempio), una volta trascorso il DIFS, anziché inviare al destinatario (C nell'esempio) la trama dati già pronta, gliene invia una di controllo molto più corta, il RTS appunto. A questo punto la stazione C (che tipicamente coincide con l'Access Point) assume un ruolo di coordinamento, e risponde con una trama di controllo di CTS. Quest'ultima viene ricevuta anche dal terminale nascosto (A nell'esempio), che da quel momento in poi, si astiene dal trasmettere, per il tempo indicato dal campo Duration presente nella trama CTS.



2.2.5 Metodo di acceso al medio CSMA/CA

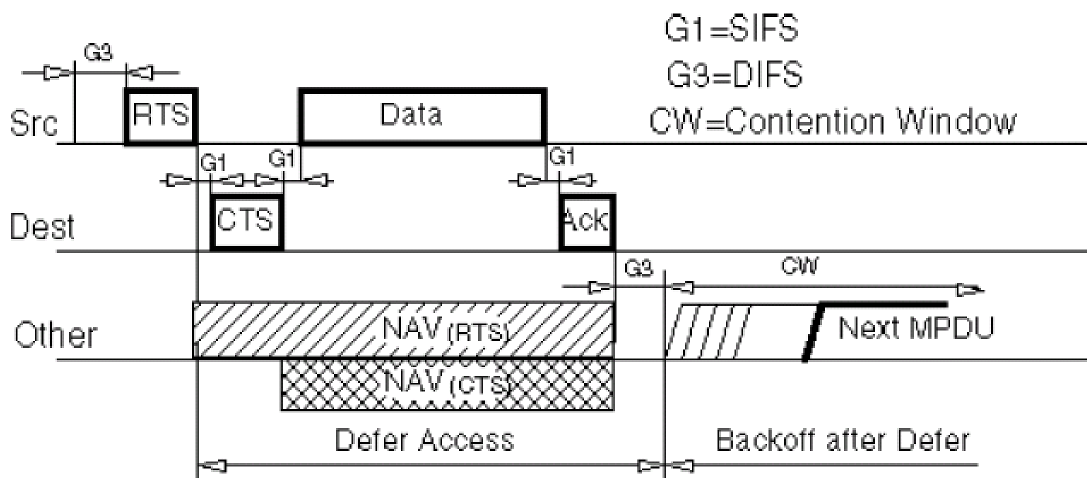
CSMA, acronimo di Carrier Sense Multiple Access (accesso multiplo tramite rilevamento della portante), è un protocollo usato nelle reti di tipologia bus per condividere la disponibilità di rete ed evitare la trasmissione contemporanea di due host.

In una LAN, due host prima di inizializzare la trasmissione di dati, devono verificare che nel cavo non vi sia la presenza della portante (Carrier Sense) e che quindi non vi siano già trasmissioni in corso. Nel caso il canale sia libero la trasmissione puo' iniziare altrimenti si aspetterà un tempo arbitrario.

CSMA non è comunque in grado di evitare conflitti. Può accadere infatti che i due host trovino la rete libera proprio nello stesso tempo e che quindi inizino entrambi la trasmissione. A causa di questo si verificano delle collisioni, ovvero i dati interferiscono tra di loro quando si incontrano nello stesso punto del cavo causando quindi la scorretta ricezione di loro stessi.

Per questo motivo, al protocollo CSMA, è stato affiancato CD (Collision Detection). In una rete quindi che utilizza questo protocollo (CSMA/CD, IEEE 802.3), ogni host che deve mandare dei dati controlla preventivamente che non ci siano trasferimenti in corso. Dopo di che prova a trasmettere. Se rileva delle collisioni interrompe subito la trasmissione, manda un segnale di disturbo a tutti in modo da segnalare la presenza di una avvenuta collisione e riprova dopo un tempo arbitrario. In caso contrario la trasmissione continua.

Oltre a CSMA/CD, un'altra evoluzione del CSMA è il CSMA/CA, (Collision Avoidance) utilizzato dalle reti wireless (IEEE 802.11b). La particolarità di questo tipo di reti è l'incapacità di garantire che tutti gli host possano raggiungersi a vicenda in ogni momento. E' quindi impossibile rilevare le collisioni e usare di conseguenza il CSMA/CD (Lo standard 802.11b infatti prevede la modalità half-duplex. Significa quindi che la operazioni di ricezione e trasmissione non possono essere eseguite contemporaneamente a differenza del CSMA/CD, IEEE 802.3 full duplex). Il CSMA/CA cerca di evitare le collisioni (Avoidance) o comunque di ridurne le possibilità. Ogni host prima di inizializzare effettivamente la trasmissione, avvisa il destinatario il quale se risponderà affermativamente (con un pacchetto ACK) darà il via alla comunicazione. In caso contrario, il mittente riproverà dopo un tempo arbitrario.



Capitolo III. WSN. Wireless Sensors Networks.

In questo capitolo vado a fare un approccio alla 6LoWPAN, e per questo, credo che sia assolutamente necessario introdurre il concetto di WSN, e anche il concetto dello standard 802.15.4 prima di affrontare como funziona 6LoWPAN.

3.1 Definizione di WSN.

I recenti progressi fatti nel campo della micro elettronica e nelle comunicacioni wireless hanno permesso lo sviluppo di sensori-node, sensori con la capacità di comunicare tra loro tramite tecnologia wireless a raggio limitato.

Una Reti di Sensori inalambriche è la è raccolta di una serie di nodi sensore "wireless", di bassi dimensioni, peso e basso costo, che si spiegano in una particolare regione. La sua funzione è fornire una infrastruttura di comunicazione "wireless" per potere fare il monitoraggio di qualunque sistema specifico, normalmente attraverso il controllo dei parametri fisici , como sono temperature, luze, etc...

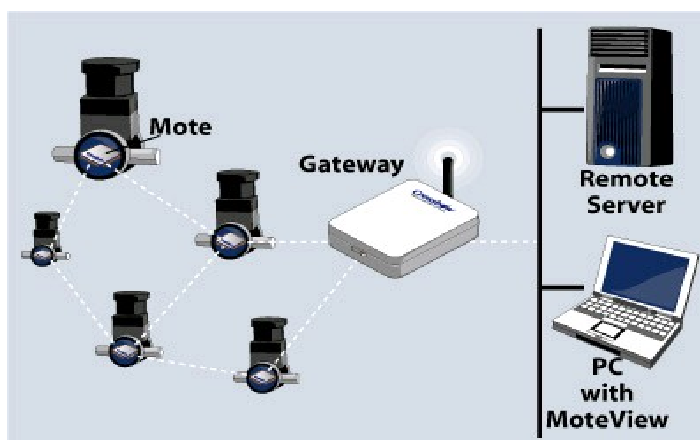
Inoltre, questi nodi sono in grado di eseguire alcune locali calcoli sui dati presi, che consente una riduzione del traffico attraverso la rete per essere dotatti di un processore e questo permette fare una riduzzione dello traffico attraverso la rete.

Le reti di sensori possono essere utilizzate in molte applicazioni; la realizzazione di queste applicazioni richiede l'uso di tecniche utilizzate nelle reti wireless ad hoc. Comunque, molti degli algoritmi usati nelle reti ad hoc non sono compatibili con i requisiti di questo tipo di reti.

I principali motivi sono:

- Il numero di nodi che compongono una rete di sensori può essere di alcuni ordini di grandezza più grande del numero di nodi in una rete ad hoc.
- I nodi sensore sono disposti con un'alta densità.
- I nodi sensore sono soggetti a guasti.
- La topologia di una rete di sensori cambia frequentemente.
- I nodi sensore usano un paradigma di comunicazione broadcast mentre la maggior parte delle reti ad hoc sono basate su una comunicazione di tipo punto-punto.
- I nodi sensore sono limitati per quanto riguarda l'alimentazione, le capacità di calcolo e la memoria.
- I nodi sensore possono non avere un identificatore globale (ID).

Le reti di sensori possono migliorare in modo significativo la qualità delle informazioni: ad esempio possono garantire una elevata fedeltà, possono fornire informazioni in tempo reale anche da ambienti ostili e possono ridurre i costi di trasmissione delle stesse informazioni.

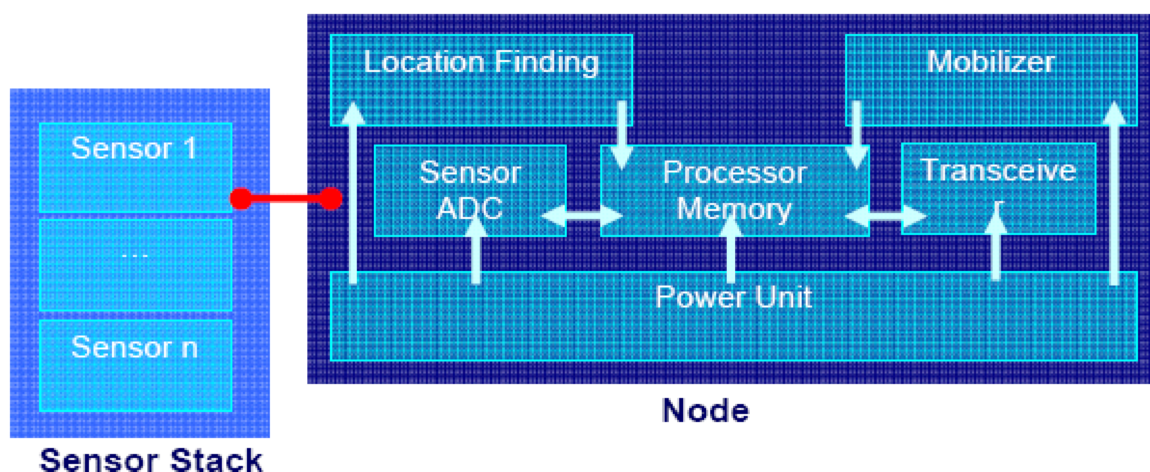


3.2 Architettura generale di un Sistema WSN.

Una wireless sensor network è solo una parte di un sistema più complesso, definito Sistema WSN. Esso, infatti, inizia dalla WSN vera e propria ma prosegue con il canale di comunicazione tra la WSN e una database di raccolta dati, che può essere un server internet, e l'interfaccia tra il database e l'utente finale.

Quindi, ha senso parlare di WSN, e soprattutto, ha senso il suo utilizzo, se non rimane confinato alla semplice dimostrazione di funzionalità della stessa, ma è in grado tramite delle interfacce opportune di interagire con l'utente finale: solo in questo modo si può ritenere che il "sensing"dell'ambiente in questione sia ragionevolmente utile.

Analizzando in dettaglio gli elementi che costituiscono una WSN, appare subito chiara la differenza tra i nodi della rete, preposti alla gestione dei sensori e al mantenimento della infrastruttura wireless della rete, e uno (o anche più) nodi, che hanno il compito di fare da collettori e trasmettere i dati ricevuti dagli altri nodi verso il server centrale.



Come si può vedere in figura, sul nodo è presente un elemento di power supply, generalmente una batteria, che fornisce l'alimentazione necessaria per tutte le funzionalità del nodo: in generale, la conservazione della carica della batteria è un elemento chiave, visto che la durata di vita dei singoli nodi e, quindi, della rete è uno dei parametri cruciali per valutare l'efficienza e la bontà di una WSN.

Cuore del sistema nodo è sicuramente il microcontrollore, che deve rispondere alle caratteristiche di basso costo, buone capacità computazionali, memoria a bordo sufficiente per garantire le prestazioni minime richieste ad una WSN, e possa consumare poca potenza,o almeno preveda degli stati di power saving, quando non viene utilizzato.

Il microcontrollore è colui, che gestisce ad alto livello tutti gli elementi presenti sul nodo:decide quando interrogare i sensori, quando trasmettere i dati e a quale nodo trasmetterli, può effettuare delle operazioni di data fusion, implementa e gestisce il protocollo di comunicazione.

Il transceiver (TRANSmitter/reCEIVER) è l'elemento di interazione del nodo con il resto dei nodi della rete, ovviamente nel range della sua portata di trasmissione: al transceiver sono affidati i compiti di ricezione e trasmissione dati e di “Sensing del canale”, ovvero deve analizzare il livello di potenza radio presente sul canale di comunicazione, per stabilire se trasmettere o meno un pacchetto dati; ad esso sono anche affidati tutti i compiti, richiesti dalla teoria delle telecomunicazioni, ovvero l'analisi della correttezza del pacchetto ricevuto, l'invio della stringa di preambolo, etc.

3.3 Sensori: interfaccia e condizionamento.

Un grande ostacolo attualmente presente, che impedisce una rapida espansione del mercato delle WSN è l'assenza di sensori integrabili, di basso consumo e basso costo, da poter alloggiare sui singoli nodi: a parte i sensori più comuni, che si trovano in commercio e che misurano grandezze “semplici”, come umidità e temperatura, non c'è molta scelta, anzi è quasi assente, per sensori che misurino grandezze più “interessanti”, come la percentuale di gas presenti in un ambiente o in grado di riconoscere e quantificare le sostanze disciolte in un liquido.

La maggiore difficoltà per arrivare all'integrazione tra la WSN e i sensori rimane il fatto, che molti sensori hanno bisogno di un "condizionamento": ovvero possono funzionare solo con determinate condizioni al contorno, facilmente riproducibili in laboratorio, ma difficilmente realizzabili su un nodo, se non a costo di abbandonare la filosofia della WSN: tra le altre, ci possono essere forti vincoli sulla temperatura di funzionamento, generalmente stabile intorno a valori elevati, anche superiori a

parecchie centinaia di gradi centigradi; oppure la necessità per i sensori di gas di mantenere un flusso continuo e costante di aria, per garantire una misurazione affidabile.

Comunque, una volta superate queste difficoltà, rimane il problema ingegneristico di interfacciare il sensore con il nodo della WSN: i sensori si dividono in due grandi categorie, quelli analogici, ovvero che forniscono in uscita la variazione di una grandezza (Tensione,Resistenza,Corrente), e quelli digitali, che forniscono direttamente un segnale digitale contenente la misura, già codificata.

Per poter garantire espansibilità e generalità alla WSN, l'interfaccia nodo-sensore deve essere almeno in grado di gestire queste due tipologie: deve quindi prevedere delle connessioni (bus), che forniscano l'alimentazione necessaria al sensore, la massa, uno o più bus dati, e per i sensori digitali, anche un bus, che fornisca il sincronismo (clock); tutto questo, insieme alla gestione del sensore (tempi di accesso, ritardo di lettura etc.) viene demandato ad una apposita elettronica, esterna al microprocessore,dal quale però, dipende e col quale comunica, tramite bus appositi.

3.4 Lo standard IEEE 802.15.4.

Queste standard define il protocolo e interconessione dei dispositivi via comunicacione in a "Personal Area Network" (PAN).

Lo standard usa carrier sense multiple access with collision avoidance (CSMA-CA), meccanismo di accesso medio e sostiene "star" così come "peer-to-peer" topologie.

LR-WPAN è una semplice rete di comunicacione low-cost che permette conetività wireless in aplicazioni con limitazioni di consumo di energia e rilassato requisiti di "throughput".

I principali obietivi di LR-WPAN sono la facilità di installazione, il trasferimento di dati affidabili, un corto raggio operazione, estremamente basso costo, e una ragionevole durata della batteria, mentre il mantenimento sia di un protocollo semplice e flessibile.

Alcune caratteristiche di LR-WPAN (in Inglese):

-Over-the-air data rates of 250 kb/s, 100kb/s, 40 kb/s, and 20 kb/s.

-Star or peer-to-peer operation.

-Allocated 16-bit short or 64-bit extended addresses.

-Optional allocation of guaranteed time slots (GTSs).

-Carrier sense multiple access with collision avoidance (CSMA-CA) channel access.

-Fully acknowledged protocol for transfer reliability.

-Low power consumption

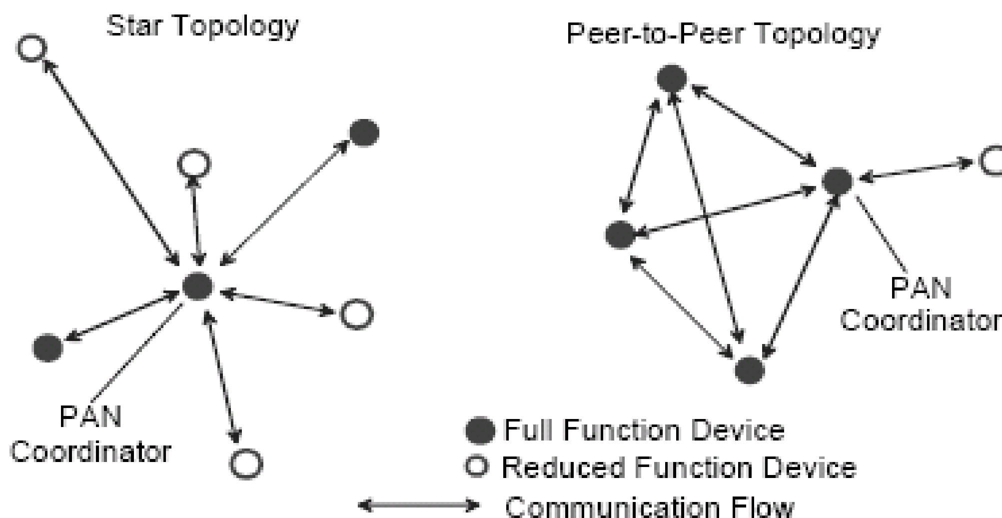
-Energy detection (ED).

-Link quality indication (LQI).

-16 channels in the 2450 MHz band, 30 channels in the 915 MHz band, and 3 channels in the 868 MHz band.

3.4.1 Topologia di Rete.

Dipende dei requisiti della applicazione, IEE 802.15.4 LR-WPAN poe operare in questi due topologie: topologia di stella o la topologia peer-to-peer.



Nella topologia **Star** tutti i dispositivi comunicano direttamente con il solo *PAN Coordinator*, centro della topologia a stella. Gli RFD rappresentano gli *end point* della rete e ogni comunicazione tra qualsivoglia *end point* della rete è mediata dal coordinatore. Questa è la topologia più semplice, consente l'impiego di protocolli poco onerosi da un punto di vista computazionale per gli RFD e risulta quindi preferibile nei casi in cui sia disponibile un solo nodo affidabile dal punto di vista energetico e computazionale e altri alimentati invece da sorgenti di potenza limitata.

La topologia di rete *peer to peer* è indicata per reti con requisiti funzionali maggiori rispetto a quelle a stella. Il *Pan Coordinator* è ancora presente, ma la comunicazione tra i nodi componenti la rete non è necessariamente mediata da tale nodo.

Questo consente di ottenere percorsi ridondanti e, di conseguenza, topologie più complesse che richiedano l'implementazione di algoritmi di *routing* più complessi rispetto a quelli adottati nelle reti a stella. Resta comunque attuale il vincolo in base al quale la comunicazione tra una coppia di RFD deve essere sempre mediata da almeno un FFD: questo permette di alleggerire la progettazione dei nodi RFD e l'algoritmo di routing che devono implementare. Tali restrizioni non sono previste invece per i dispositivi FFD.

3.4.2 Livello fisico (PHY).

Il livello fisico dello standard 802.15.4, PHY, è responsabile delle operazioni che permettono la comunicazione tra i nodi della rete. La maggior parte delle operazioni vengono implementate direttamente nell'hardware del dispositivo; lo standard non esprime tuttavia regole in merito, lasciando piena libertà ai costruttori relativamente all'implementazione del PHY. Le operazioni richieste al livello firmware sono:

-Spegnimento e attivazione del il ricetrasmettitore.

-Misurare la quantità di energia presente nel canale, Energy Detection (ED), prima di ogni comunicazione, per stabilire se operare su una determinata frequenza o appurare la presenza di disturbi nel canale che ne impedirebbero la trasmissione.

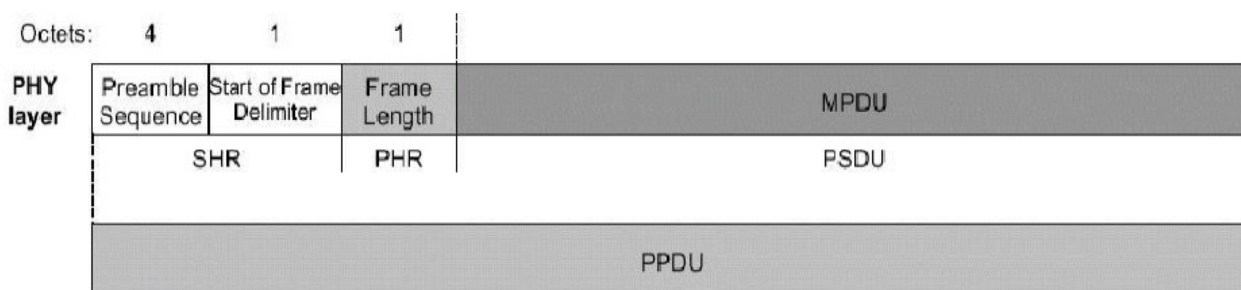
-Misurare la qualità del collegamento per mezzo di un opportuno parametro chiamato LQI, Link Quality Indicator, all'atto di ricezione di un pacchetto.

-Verificare la presenza di segnale sul canale wireless per evitare collisioni tramite il CCA, Clear Channel Assessment.

-Selezionare il canale migliore su cui effettuare la comunicazione

-Effettuare la ricezione e la trasmissione dei dati.

Due dispositivi che vogliano scambiarsi delle informazioni al livello fisico, devono formattare opportunamente i dati raggruppandoli in sequenza di byte.



A questo livello, il pacchetto finale che viene trasmesso da un dispositivo è noto come PPDU, ossia **PHY Protocol Data Unit**, ed è così strutturato:

Il pacchetto inizia sempre con:

-Un **preambolo** di quattro byte, composto da trentadue zeri, utilizzato per la sincronizzazione.

-Un byte, che assume il valore 0xA7, utilizzato come delimitatore di **inizio frame** che rappresenta l'inizio vero e proprio del pacchetto.

-Un byte che indica la lunghezza del **PHY Service Data Unit** (PSDU) che segue. La lunghezza è espressa dai sette bit meno significativi, essendo l'ottavo riservato.

-Il PSDU vero e proprio che coincide con il **MAC Protocol Data Unit**, (MPDU) ossia con il pacchetto preparato dal livello superiore.

3.5 *Lo sviluppo di IPv6 over IEEE 802.15.4*. 6LoWPAN.

6LoWPAN è il nome di un Working Group all'interno dell'IETF, Internet Engineering Task Force, il cui scopo è definire le specifiche per la trasmissione di pacchetti IPv6 su reti IEEE 802.15.4.

l'802.15.4 prevede quattro tipi di frame: *beacon*, *command*, *data* e *acknowledgement*. I pacchetti IPv6 devono essere incapsulati all'interno dei data frame. È consigliabile, ai fini di una trasmissione affidabile dei pacchetti, che la trasmissione di questi frame avvenga richiedendo l'invio di un *acknowledgement* frame in risposta all'avvenuta ricezione.

L'uso di frame beacon e MAC Command non è invece richiesto per trasportare pacchetti IPv6. Utilizzando reti non beacon enabled,i data frame sono inviati attraverso il canale secondo una metodologia di contesa utilizzando il protocollo CSMA/CA per la trasmissione. In reti così configurate, i beacon frame non vengono utilizzati per la sincronizzazione, ma possono essere utili per gestire fasi quale la neighbor discovery, ND, sfruttando indirizzi link-local, e l'associazione/disassoci azione di un dispositivo.

3.5.1 Indirizzi in 6LoWPAN

Questo è solo un breve riassunto di come gli indirizzi vengono trattati in 6LoWPAN siguendo la 802.15.4 standard.

L'802.15.4 definisce due tipologie di indirizzi: **indirizzi estesi IEEE EUI-64 bit**, univoci globalmente, o **indirizzi short a 16 bit**, univoci soltanto all'interno della PAN. Entrambi i tipi sono supportati dal 6LoWPAN.

Per utilizzare indirizzi short a livello di rete, dato che essi sono disponibili soltanto dopo l'assegnamento da parte del coordinatore della PAN e la loro validità è limitata unicamente alla validità dell'associazione, il 6LoWPAN impone che l'utilizzo venga effettuato sotto precise condizioni ed entro certi limiti.

In 6LoWPAN si assume che una PAN sia mappata all'interno di uno specifico link IPv6, questo comporta l'utilizzo di un prefisso comune per i dispositivi della rete.

Il 6LoWPAN permette l'integrazione fra i due livelli qualora i frame broadcast 802.15.4 siano ascoltati soltanto da dispositivi. Questo comporta che soltanto i dispositivi che condividono un determinato prefisso di rete, associati sullo stesso link potranno ricevere i messaggi di questo tipo, composti come segue:

-I pacchetti IPv6 multicast devono essere incapsulati in frame broadcast IEEE 802.15.4;

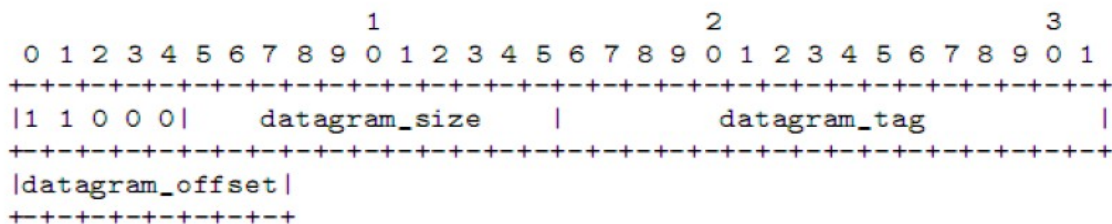
-Un identificatore di PAN è incluso nel frame IEEE e coincide con il PAN ID della comunicazione sul link in questione;

-Un indirizzo short di destinazione è incluso nel frame e deve essere l'indirizzo **0xFFFF**.

3.5.2 Composizione Frammentazion Header 6LoWPAN.

Un meccanismo di frammentazione dei pacchetti è indispensabile per poter incapsulare pacchetti IPv6, lunghi fino a 1280 byte, all'interno di pacchetti 802.15.4 il cui payload disponibile varia dagli 81 ai 102 byte.

Dopo i primi 5 bit di prefisso per il dispatch, seguono un campo di 11 byte che rappresenta il campo Datagram Size e un campo 16 bit Denominato Datagram Tag.



Il campo Datagram Size indica la grandezza del pacchetto originario non frammentato escludendo gli header 6LoWPAN opzionali.

Questo campo deve essere presente in ogni frammento creato per poter allocare un buffer di memoria della grandezza corretta. Il campo Datagram Tag è anch'esso ripetuto per tutti i pacchetti ed è distintivo dei frammenti di uno stesso pacchetto.

Esiste, infine, un ultimo campo nell'header di frammentazione, denominato datagram offset che non compare esclusivamente nel primo frammento. Esso indica, con un incremento di 8 bit, l'offset del frammento dall'inizio del payload.

La ricomposizione di un frame a destinazione deve avvenire entro 60 secondi19 dalla ricezione del primo frammento di un pacchetto, frammento riconoscibile dal valore del dispatch e identificabile in base al valore del datagram_tag.
In caso la procedura non si completi entro 60 secondi, tutti i frammenti ricevuti vengono scartati.

L'introduzione a specifica delle funzionalità di frammentazione ha la sua principale ragione nella conformità con il protocollo IPv6. In generale, tuttavia, è statisticamente verificabile come le applicazioni specifiche per reti 802.15.4 generino pacchetti di dimensioni relativamente ridotte in un'ottica di risparmio energetico e di risorse trasmissive.

Per ridurre il numero di informazioni necessarie al protocollo IPv6 pertanto, si tende a prediligere un meccanismo di compressione degli header che permetta di incapsulare tutte le informazioni in un unico pacchetto 802.15.4 senza sfruttare i meccanismi di frammentazione descritti sopra.

Principali possibili Heads in 6LoWPAN, Il Mesh Header, e il Broadcast Header.

Il Mesh Header:

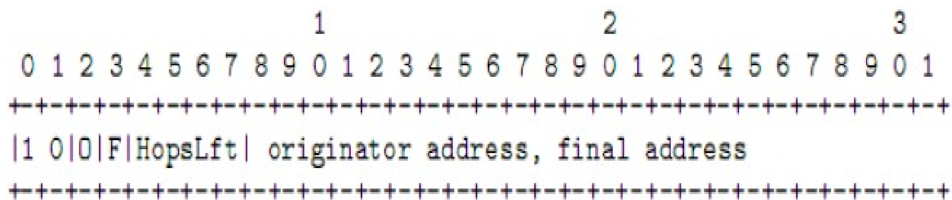
L'header è composto da un dispatch di 8 bits:

BITS 0, 1 : Prefisso costante, 10, che identifica il tipo di header Bit.

BIT 2 : Flag settato a 0 per indica che l'indirizzo del mittente, contenuto nel campo dell'header, è in versione EUI-64, altrimenti 1 se presente in formato short .

BIT 3 : Flag settato a 0 per indica che l'indirizzo del destinatario, contenuto nel campo dell'header, è in versione EUI-64, altrimenti 1 se presente in formato short Bit.

BITS 4,5,6,7 : Usati per indicare il numero di salti ancora previsti per arrivare al destinatario. Ad ogni hop questo campo viene decrementato di uno. Se si arriva al valore 0 il pacchetto viene scartato. Previsti massimo 16 hop.



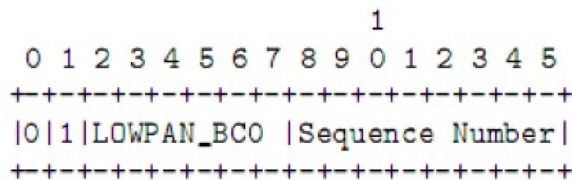
L'header non specifica quale protocollo di routing mesh adottare ma rappresenta soltanto un supporto per codificare le informazioni necessarie ad un qualunque protocollo di routing mesh.

<u>Il Broadcast Header:</u>

Un header opzionale usato spesso in combinazione con l'header precedente, è quello riservato alle comunicazioni broadcast.

BITS 0,1: Prefisso costante, 01, che identifica il tipo di header.

BITS 2,3,4,5,6,7: Broadcast Dispatch.



3.5.3 Compressione Header 6LoWPAN.

È stato già evidenziato in precedenza come, nel caso più sfavorevole, in un frame 802.15.4 siano disponibili per i livelli superiori soltanto 81 byte.

Considerando che l'header IPv6 occupa da solo 40 byte, rimarrebbero soltanto 41 byte per i dati dei livelli superiori.
Nel caso venisse usato anche il protocollo UDP dovremmo considerare un ulteriore header di 8 byte e, di conseguenza, soltanto 33 byte per il livello applicativo.

È evidente dunque, che un meccanismo di compressione degli header risulti indispensabile per potere utilizzare in maniera razionale il protocollo IPv6 su reti IEEE 802.15.4.
Il meccanismo, adottato allo scopo di eliminare le informazioni ridondanti, prevede un'integrazione tra il livello 3, 6LoWPAN, e il livello 2, MAC 802.15.4, della pila protocollare.

6LoWPAN attualmente definisce inoltre un meccanismo di compressione per gli header IPv6 e UDP, mentre non è stato formalizzato alcun meccanismo per i protocolli ICMP e TCP.

In questa Tesi non vado a fare la descrizzione del meccanismo di compresione, questo è solo un breve orientamento al suo funzionamento.

3.5.4 Principali soluzioni commerciali sul 6LoWPAN.

3.5.4.1 Introduzzione e descrizione di Nodo-Sensore.

Un node sensore è un elemento computazionale con la capacità di procesamento, di memoria, con interfacce di comunicacione e composto di microsistemi di sensori. Cumunque, il Hardware basico di un dispositivo è costituito da:

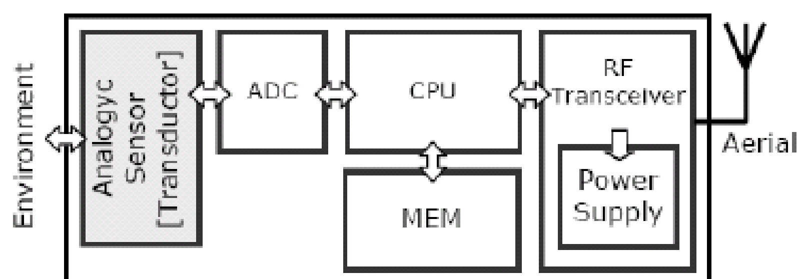
Un transcettore, che permete l'invio e recezione di messagi tra il node e l'altri elementi della rete.

Un processore, che fa il controllo dell'activitá del nodo.

La memoria, per la memorizzazione dei dati.

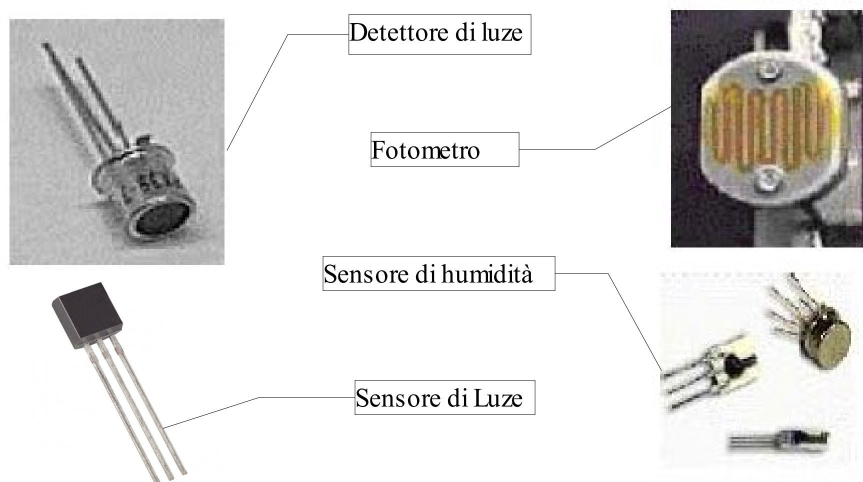
Il generatore, che è formato da Batterie.

I sensori, che sono i responsabili da racoltere i dati ambientali e fare la conversione di questi dati di analogico a digitale.



Questi sensori-node sono piccoli componenti wireless di procesato, che utilizzano la tenologia MEMS. Sono formati da unità di sensori che posono essere da diverso tipo (acustici, sismici, infrarossi, meccanici, di calore, di temperatura, di radiazioni ...).

Diversi tipe di sensori comerciali:



Attualmente, si produrrono sensori a basso costo, e con piccoli dimensioni. È fundamentale risaltare che tutti questi nodi hanno poca capacità di memoria e procesamento, e per questo i microprocesori devono essere di basso consumo.

Comunque, i aplicazioni devono facilitare un controllo di guasto energetico. In queste senso si hanno sviluppato elementi Hardware e Software con l'scopo di fare un minimo guasto energetico, e cosi, raggiungere un tempo di vita maggiore.

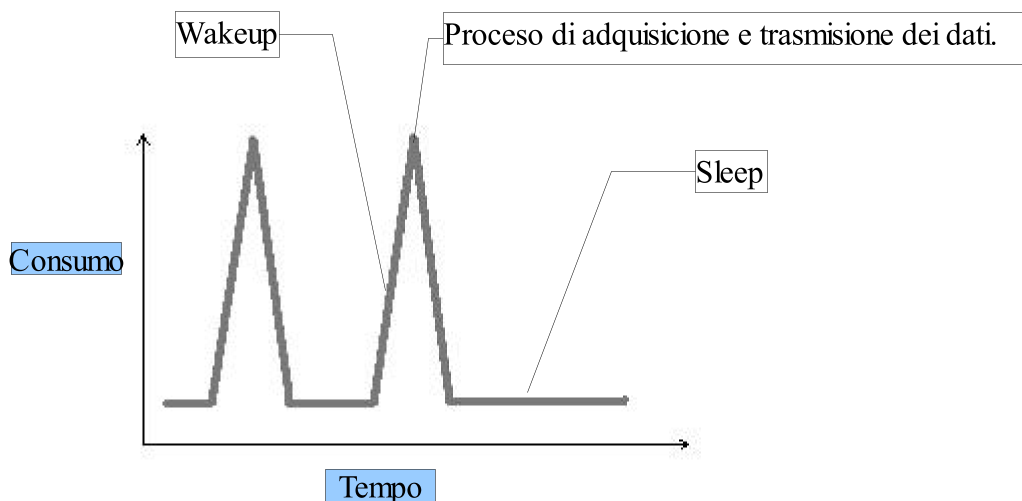
I nodi, accordo al suo modo di funzionamento, possono rimanere in tre stati fundamentali:

1.Sleep: - In queste stato il nodo e inattivo. Lo scopo è ottenere che quasi tutto il tempo il nodo sia n queste stato, e como risultato, il suo consumo sia minimo.

2.Wakeup: - Queste è l'stato dove il nodo sveglia, e diventa a un stato attivo. Queste stato sucede cuando arriva qualcun stimulo o interruzzione programata. Lo scopo è che il nodo non rimanga in queste stato molto tempo per consumire meno energia.

3.Active: - In queste stato è dove il nodo fa il laboro di adquisito, procesato e trasmisione di dati. Como nel wakeup, deve rimanere in queste stato il tempo minimo possibile.

In questa image possimo vedere il consumo di un nodo secondo l'stato nel che si trove.



3.5.4.2 - Diverse opzione comerciali.

Esistono nel mercato diversi opzioni nel momento da fare l'elenco di qual è il tipo di nodo ho bisogno. Qua vado a mostrare una picola parte degli opzioni che hanno nel mercato.

I modeli piu populari che sono nel mercato adesso, sono Telos e Micas, ma in questa Tesi si ha laborato con la serie NanoStack di Sensiode. Prima, vado a fare una descrizione dei modeli Micas e Telos, e dopo in profundità vado a describire il DevKit di NanoStack.

Il modelo Telos.

-I modeli TPR2400 e TPR2420 sono di codize aperto, svilupati e pubblicati per la Universitá di California, Berkeley.

Ogni uno di questi modeli ha un processore incorporato , microcontrollore TI MSP430, con una interfacce USB per la sua programacione, 4.4 GHz di radio, una antena PCB, e diversi unitati.

Il modelo TPR2420 è come il modelo TPR2400, salvo che il primo contiene una unità sensore di ambiente preinstallata.

TELOS TPR2420

TELOS strumenti rendono ideale per la lavorazione in laboratorio. Sui principali aplicazioni sono:

- La sua utilizacione nel sviluppo di aplicazioni di ricerca di bassa potenza.
- La sua utilizacione per la sperimentazione in Wireless Sensor Networks.

Queste modelo ottene un basso consumo di energia, che permete una lunga vitta alle baterie, in piu di avere un tempo minimo per restare nel stato de Wakue-up, un altro scopo nella strategia di basso consumo.

Per la programacione o per la comunicacioni, sono colegati al PC per il porto USB.

Anche si posono colegare dispositivi per i due connettori di espansione, como per essempio sensori analogiche, digitali e displays LCD dopo di essere configurato.

Inoltre, le sue caratteristiche principali sono :

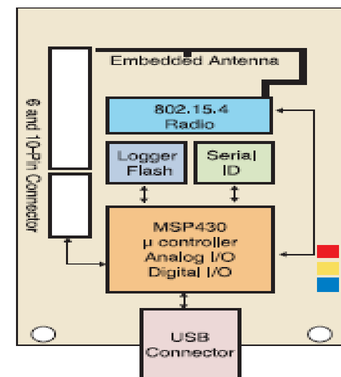
• Trasmissione RF secondo IEEE 802.15.4/ZigBee.

• Banda di frequenza da 2,4 GHz a 2,4835 ELOC .

• Trasferimento dati 250 kbps.

• Antena integrata.

• MICRO MSP430-controllore a 8Mhz con 10kB di RAM.



Schema a blocchi del modello TPR2420

• Il basso consumo.

• 1MB flash esterno per la memorizzazione dei dati.

Il modelo Micas.

Qua vado a descrivere i principali caratteristiche dei principali node di questa familia: MicaZ, Mica2 e Mica2Dot. Tutti loro si programano con un “Devboard” MIB510, queste “Devboard” fa di interfacce tra il PC e i nodi per il porto serie, e questo permete la programacione dei dispositivi con un procesore (ISP) Atmega16L.

Il codice si scarica al ISP traverso il porto serie RS-232, e il ISP fa la programacione del codice.

MIB510

MicaZ:

- Il MicaZ è “Embedded” con un transceiver ZigBee di radio frecuenza che labora nella banda di 2400 MHz – 2483.5 Mhz e utiliza una modulazione digitale Direct Sequence Spread Spectrum (DSSS), ha una velocità di 250 Kbps e sicurità per Hardware (AES-128). In connettore supporta le interfacce di espansione di input analogici/ output (I / O) digitali.

In piu, usa il chipcon CC2420, basso la norma dello standard IEEE 802.15.4 e un microcontrollore Atmega128L per la programacione dei nodi.

MicaZ

Mica2:

I nodi Mica2, si usano in reti di sensori di bassa potenzia. Migliorano i caratteristichi dello Mica originale.

La progettazione di queste tipo di sensori è specifiche per reti di sensori integrati:

- Diversi frecuenzi di trasmissione di amplio rango.
- Piu di un anno di batteria atraverso il mode sleep.
- Permetteno riprogramazione wireless.
- Vasta gamma di sensori compatibili, di luze, temperature, presione, acelerazione …

I diversi aplicazioni dove si utilizano questi nodi sono, principalmente, le WSN e la securità, monitoraggio ambientale o i reti wireless da grande scala.

I Mica2 si dividono in tre prinzipali modeli, dipendendo del modo di uso:

MPR400 (915MHz), MPR410 (433MHz), MPR420 (315MHz).

Questi nodi usano il Chipcon CC1000, con radio modulata in FSK. Tutti i modeli usano un potente microcontrollore Atmega128L e una radio frecuezia di amplio rango.

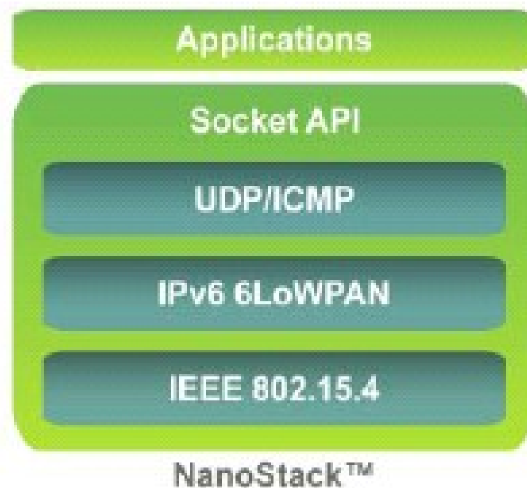
Mica2

3.5.4.3 Sensinode. La gamma Nano series.

Il laboro di questa Tesi si ha realizzato con la gamma di Nodi Nano series, della società Sensinode.

Questa società si trova a Finlandia, e fa la produzzioni di nodi che si basano sulla tecnologia pioniere 6LoWPAN, che è anche basati su IP versione 6. Il uso degli standars IEEE 802.15.4, Ip e ZigBee, danno a sui prodotti una interoperabilità globale e un tempo di vitta alto. In piu, sui nodi-sensori permetteno realizzare una vasta gamma di diversi misuri degli fenomeni del medio, cosi como per controllare gli oggetti, le persone e le zone in tempo reale .

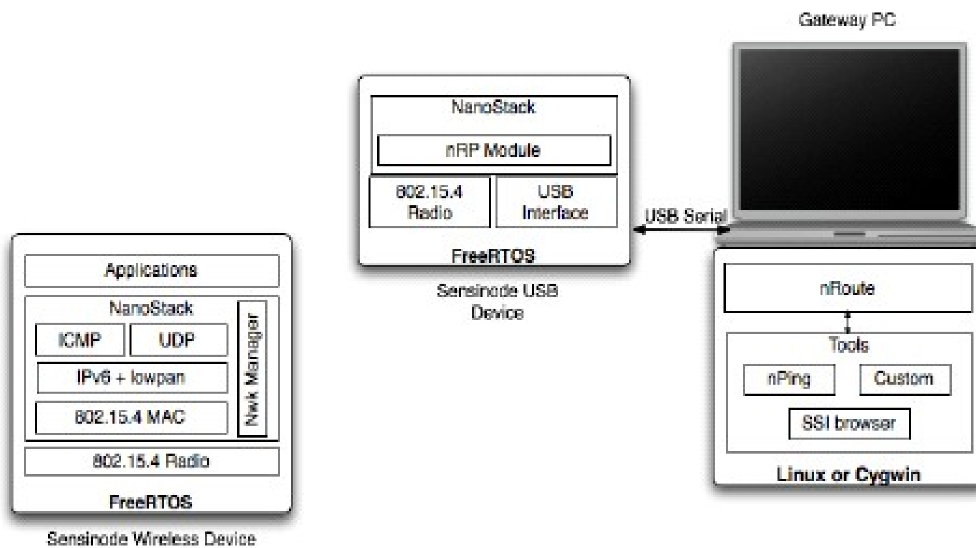
La caratteristica principale dei nodi densori di Sensinode e il uso dello stack di protocolli di NanoStack:



NanoStack include l'acceso a i dati intercambiati tra comunicacioni traverso di un facile uso della interfacce Socket. Attualmente, questa interfacce è ampiamente utilizzato nelli comunicacioni tra computers.

Soprattuto è utilizato negli sistemi POSIX, è cosi si diventa in un instrumento molto utile per i sistemi embedded, come sensore di nodi. La API di NanoStack è molto similare alla API di POSIX, e aggiunge funzionalità per la gestione della memoria per raggiungere flessibile operazioni con buffers.

Queste stack di protocolli supporta sia l'ambiente Linux como Windows (CYGWIN) utilizzando strumenti open source (gcc, make, sdcc …). A continuazione si mostrano nella image i componenti interni della architettura NanoStack.



NanoStack è costruito in modo stabile e portatile sistema operativo in tempo reale chiamato FreeRTOS. Queste ha un microkernel con un programatore, un microcontrollore, memoria, ed i semafori, all'interno di un sistema funzionale temporaneamente. Inoltre, è portatile ad un grande varietà di architetture e compilatori.

Un'altra caratteristiche di NanoStack è che corre como un simple Task nel interno di FreeRTOS, e cosi permette ridurre il uso di RAM e permette un controllo di flusso efficace. In piu, il uso dello Stack del protocollo si ve simplificatto perche non si usano chiamate di funzioni tra loro. Inoltre, i buffer sono distribuiti lungo una sola coda , e cosi la applicazione dello usuario non si blocca durante la operazione dello stack del protocollo.

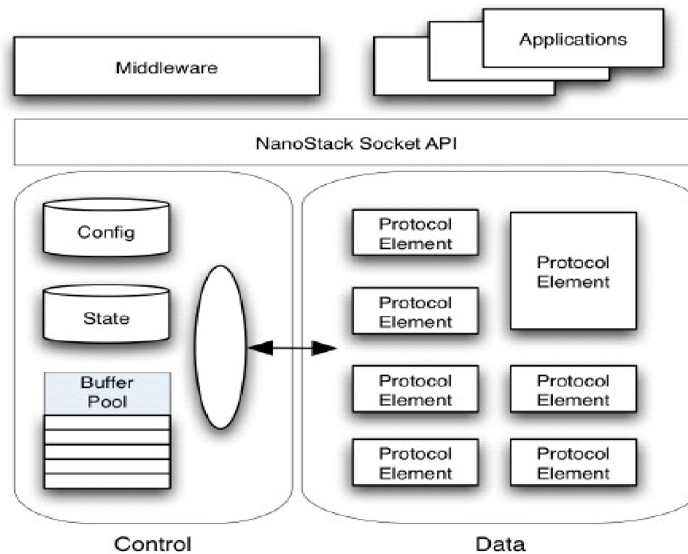


Figure 4: NanoStack internal components.

Nella architettura di NanoStack si utilizano protocolli prima descritti come IEEE 802.15.4, specificazione di come sviluppare diversi applicazioni Wireless, o 6LoWPAN, specificazione IETF che permette l'uso degli standars IPv6 e UDP sul reti wireless IEEE 802.15.4.

Ma per caratterizare bene lo stack di protocolli di sensinode, doviamo anche parlare degli specifichi protocolli usati per Sensinode nella gamma NanoStack.

- Protocollo SSI:

  Permette l'accesso ai dati dei sensori.

- NanoMesh:

  Metodo utilizzato per sensinode per fare che una rete 6LoWPAN possa proporzionare copertura Multi – Hop. NanoMesh utilizza la caratteristiche mesh-header di 6LoWPAN.

- nRP:

  Usatto nella comunicacione tra PC host e un dispositivo “serie” e cosi permettere l'accesso dello computer alla rete di sensori locale.

3.6 Principali Applicazioni di WSN e 6LoWPAN.

Como gia si ha parlato in queste documento, le rete WSN si hanno sviluppatto molto negli ultimi anni, sopra tutto per l'sviluppo di tennologie come ZigBee o 6LoWPAN, che permetteno un consumo energetico minimo e la possibilità di aprire alle WSN la porta della connetività totale con la compatibilità con IPv6.

Questa caratteristica fa che queste tipo di reti si possano utilizare nei più disparati settori;

Come per essempio nel settore militare; nella sorveglienza dei campi di battaglia.

Nel settore Civele; nella "Domotica", nel controlo all'interno di veicoli, nella Geolocalizzazione, nella Bieomedica o nella HCI.

Essistono diverse mode di uso principali di una WSN diendendo di che funzione devono realizare, i principali mode sono:

-Monitorizzazione Sincrona: Molte node organizzati e sincronizati che sincronamente fanno measuri e inviano i dati.

-Monitorizzazione in base a eventi: I nodi non inviano continuamente informazione, se no che solo lo fanno nel caso di che suceda qualcosa.

-Monitorizzazione per localizare un oggetto: Aplicazione per fare il controlo di oggetti che stano in una regine determinata.

-Hybrid reti: Quando una applicazione ha caratteristiche delli altri modi.

Capitolo IV. Caratterizzazione di un Sistema per la misurazione di parametri ambientali interni.

In queste capitolo vado a fare, prima, le descrizione di tutti gli elementi che compongono il DevKit K210, ussato per fare le prove in questa Tesi, secondo, vado a spiegare il software sviluppato per la recezione e la trasmisione dei dati tra i sensori, e per ultimo, vado a mostrare le prove realizatte per comprobare il comportamento del software sviluppato.

4.1 Il 6LoWPAN DevKit K210 of Sensinode NanoSeries.

Il Sensinode K210 6LoWPAN Devkit è significato per la ricerca e l'ingegneria, e fornisce una serie completa di strumenti software e hardware per lo sviluppo del wireless 6lowpan rilevamento utilizzando la tecnologia basata su IP da NanoStack.

Lo sviluppo kit comprende un flessibile banco laboratorio Devboard, che danno accesso ai segnali,

funzioni di programmazione e il debug.

Una selezione di 8 Sensinode Nano Serie sono inclusi i prodotti, permettendo la creazione di vari sensori wireless topologie di rete con NanoRouter ™ USB "sticks" e wireless NanoSensors ™.

Composto di:

- 6LoWPAN software development kit (CD-ROM)
- FreeRTOS and full toolchain distributions
- NanoStack™ Pro license
- Sensinode portal and engineering support
- Full PDF manuals and quick start guide
- 1 Devboard D210 for debugging & programming
- 2 RadioCrafts RC2301AT modules
- Motherboard adapter with pins for D210 use
- 2 NanoRouter™ N601 USB Sticks
- 4 NanoSensor™ N711 Sensor Nodes
- 4 battery sets for N711
- 1 USB cable
- Carrying case

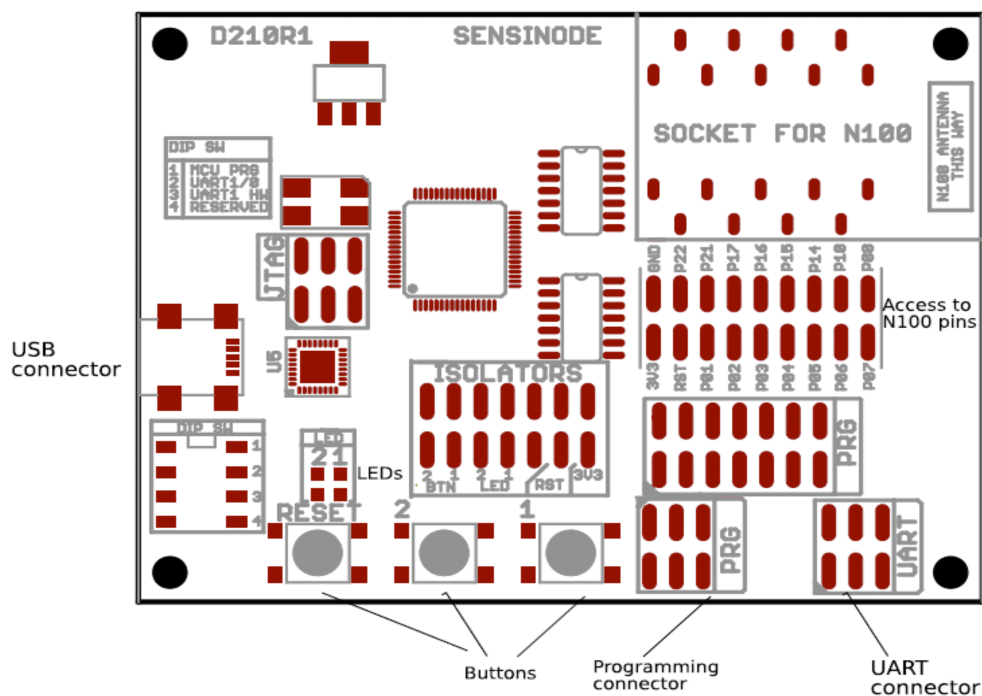
I principali caratteristiche sono:

- Ultra-compact 12.7 x 24.5 x 2.5 mm modules
- Texas Instruments CC2431 SoC radio chip
- 32 MHz 8051 microcontroller core
- Real-time 32 kHz clock
- Hardware positioning engine
- Out-of-the-box 6LoWPAN networking
- PC tools and examples included
- Linux, Cygwin and Windows support
- A complete development platform based on open-source, open-standard tools

Descrizzione del Devboard D210.

Caratteristiche elettriche:

Symbol	Parameter	Min	Max	Units
Vcc5V	5V Supply Voltage	GND-0.3	6.0	V
Vcc3V3	3V3 supply voltage	GND-0.3	3.6	V
	Bus IO Input Voltage	GND-0.3	Vcc+0.3	V
	Bus IO Output Voltage	GND-0.3	Vcc+0.3	V
	Current into Any IO-Pin	-10	+10	mA



Power Supply:
Il D210 Devboard è alimentato dal bus USB.
Il modulo connettore N100 può essere isolato dalla rete elettrica mediante il 3V3 isolatore su Devboard. È opportuno notare che, anche quando il modulo N100 è collegato al Devboard, l'alimentazione USB del Devboard dovrebbe essere usata.

Isolator Jumpers:
L' "Isolator Jumper Block" consente al N100 di collegarsi al power, reset, buttons e LEDs sul Devboard. Per impostazione predefinita tutti i jumpers devono essere installati per il normale utilizzo del modulo.

USB:
Il port USB sul Devboard può essere utilizzato per trasferire il firmware del Devboard, per alimentare il modulo N100, per programmare i moduli Sensinode Nxxx Series e per fornire una console seriale per il debug. Per trasferire il Devboard firmware, il DIP-switch deve essere impostato sulla posizione MCU_PROG.

Reset:
Il pulsante di reset sul Devboard può essere utilizzato per resettare la Devboard. Se il “jumper” isolatore di reset è installato questo pulsante può anche resettare il N100 module.

UART system:
Il USB basato in serial port sul Sensinode D210 Devboard può essere utilizzato per molteplici scopi. La funzione UART è stata selezionata mediante l'interruttore DIP. Nota che quando la programmazione di un modulo sul Devboard o un dispositivo collegato al PROG pin, non c'e alcuna necessità di modificare le impostazioni del DIP.

La programmazione dei comandi vengono rilevati automaticamente sull'interfaccia USB.
Per il suo normale funzionamento, il DIP viene impostato su {ON, ON, OFF, OFF} il quale “routes” o il UART1 dal connettore del NanoModule o il UART pin header all'interfaccia del seriale USB.

Altre opzioni permettono il controllo di flusso hardware (RTS / CTS) di essere instradato verso l'interfaccia seriale, USB, o verso UART0.
L'opzione di programmazione Devboard MCU (tutte OFF) deve essere utilizzata solo quando avviene l'aggiornamento del firmware sul Devboard.

DIP SW 1	DIP SW 2	DIP SW 3	DIP SW 4	Function
ON	**ON**	**OFF**	**X**	**UART 1 to USB + HW flow control (Default)**
ON	ON	ON	X	UART 1 to USB
ON	OFF	ON	X	UART 0 to USB
OFF	OFF	OFF	X	Devboard MCU programming

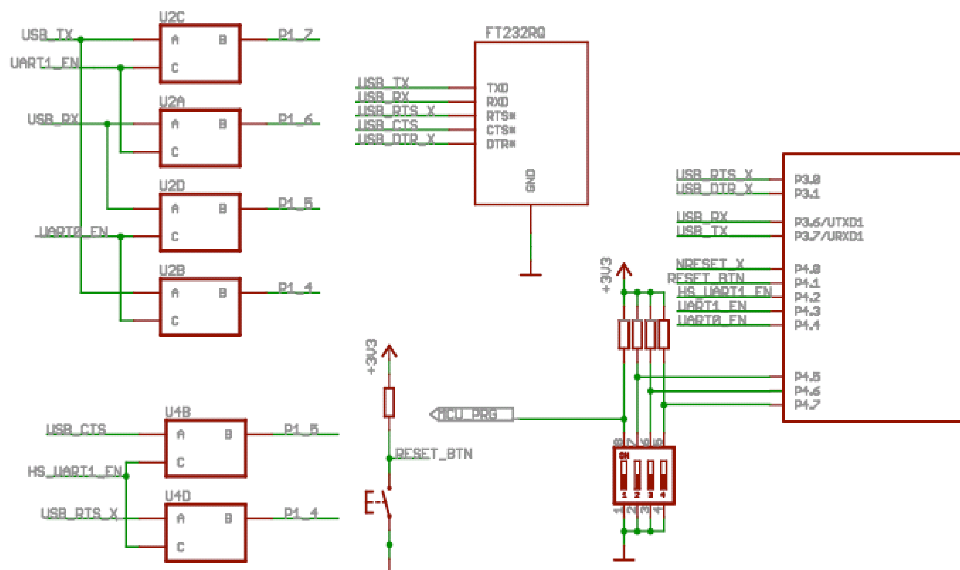


Figure 2: D210 UART routing

Descrizzione del NanoSensor N711.

Il NanoSensor N711 è un essempio di un sensore bassato nella tennologia IPv6. Il N711 include un sensore di temperatura, un sensore di luce, due LEDs e due pulsanti. Il modulo di Radio del N711 e un CC2431, chi oferta 250Kbps di Data rate, e la possibilità di ad-hoc comunications con la utilizzazione di diverse topologie. Il CC2431 è programmabile a traverso dello Sensinode Devboard.

Caratteristiche principali:

• AA battery holder integrated
• 2 LEDs and 2 buttons
• Temperature sensor
• Light sensor
• Open device with programming tools for user firmware and NanoStack™ support
• Programmable through the Sensinode Devboard
• Integrated RadioCrafts RC2301AT module
• Powerful TI CC2431 32 MHz single-cycle low power 8051 MCU
• 2.4 GHz IEEE 802.15.4 compliant RF-transceiver with 250kbps data rate
• 128kB programmable FLASH, 8kB SRAM
• Integrated positioning engine
• Integrated chip-antenna
• Solder pads for common signals for prototyping
• PIN-headers for programming and UARTs
• RoHS compatible
• Meets CE, FCC and ETSI requirements

Condizioni elettriche racomendati:

PARAMETER	SYMBOL	CONDITIONS	MIN	TYP	MAX	UNIT
Supply Voltage	V_{cc}		2.5		3.6	V
Operational temperature	$T_{operational}$		0		40	DEG C

Descrizzione dello Circuito:

I componenti dello N711 sono i sensori U1 e U2, i pulsanti S1 e S2 e i LEDs D1 e D2.

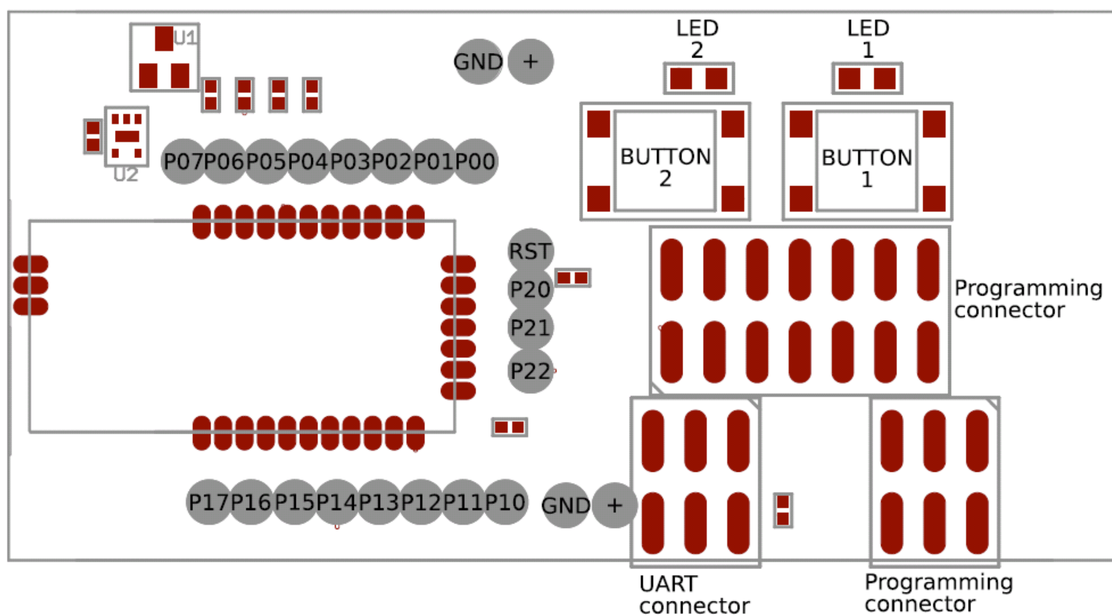


Figure 1: N711

Programmacione dello N711.

Il N711 NanoSensor si programma utilizando il D210 di Sensinode. Un conettore di 6-pin nello N711 si connetta a traverso un cavo al conettore dello D210.

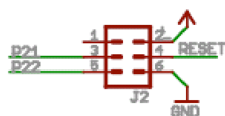


Figure 7: N711 6-pin programming connector

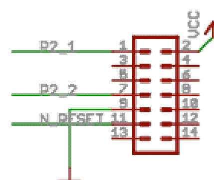


Figure 6: N711 14-pin programming connector

4.2 - Laboro di configurazione dello Stack di Sensinode.

Prima della realizzazione del codice, sono stato bisogno di un tempo di familiarizzazione con la programmacione di node-sensori in FreeRTOS.
Il lavoro inizia con la lettura di documenti per conoscere sopra che cosa sono laborando e per acquisire i necessari conoscimenti per capire il funzionamento di una rete di sensori wireless 6LoWPAN.

Dopo capire il funzionamento di 6LoWPAN, e prima d'iniziare il laboro di programacione ho dovuto fare tutta la configurazione nel computer dello stack di sensinode. Ho fatto l'elenco di laborare in Windows Vista con il emulatore di POSIX Cygwin. Anche ho cercatto sul funzionamento di FreeRTOS, perche è como labora l'stack di Sensinode.

Tutti i file della installazione sono specificati e ubicati nel CD del DevKit di Sensinode.

All'inizio l'elenco di Cygwin non ha suposto nessun problema, perche Cygwin funziona bene con i programmi piccoli, ma doppo con un software un poco piu complesso come il software che si mostra in questa Tesi, Cygwin como è un emulatore, fa la programmazione dei sensori molto lenta, e questo è un impedimento per provare il codice, perche per provare il codice si deve fare sempre la programmazione del sensore.

Per questa raggione, per il prossimo laboro sopra queste mote, penso potrevve essere meglio utilizzare un SO di tipo POSIX come LINUX, invece di un emulatore come CYGWIN.

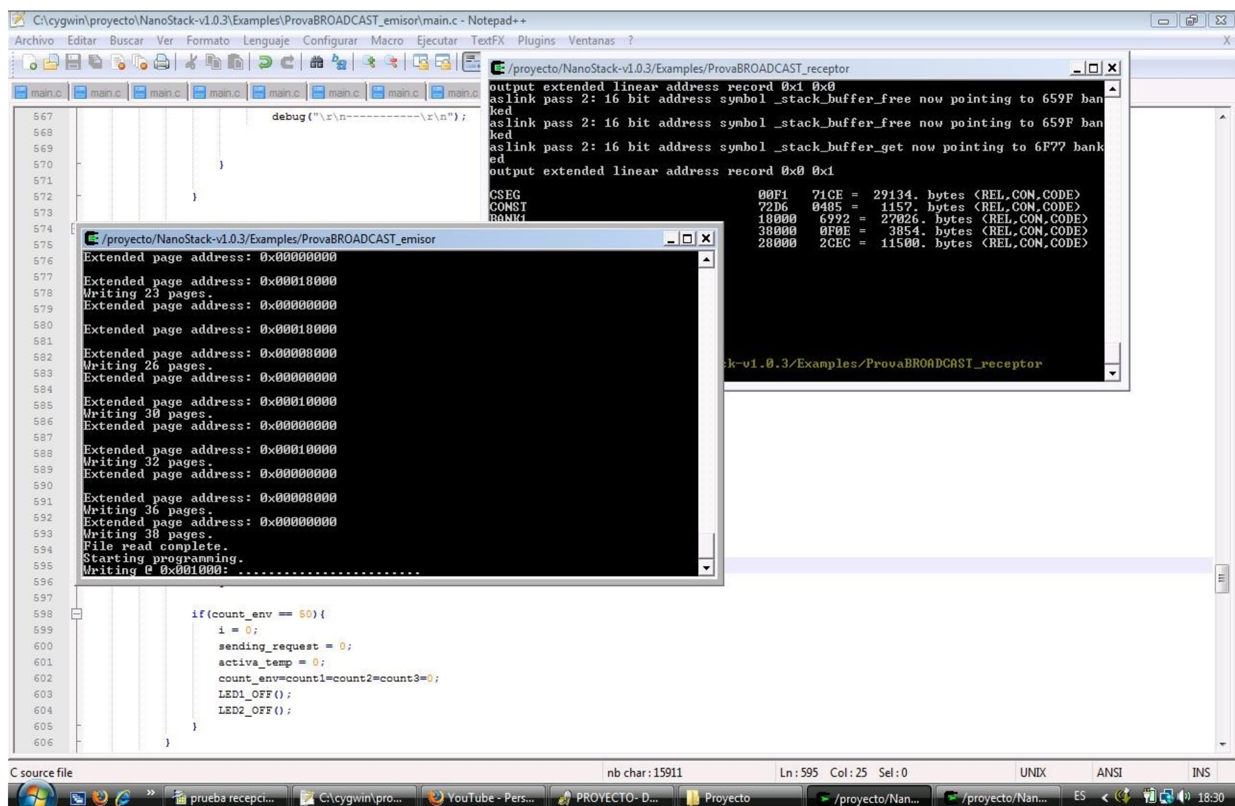
Per potere fare la prima programazione si deve prima configurare il Devboard. Tutta la spiegazione di como fare questa tare c'e scrita nel file *NanoStack HW configuration* e anche nel file *Steps for runing Programs*. Questi due file sono inclussi el cd dove sono i due programmi, nel direttorio ***/Readme***

I primi programmi fatti sono sopra tutto per iniziare a capire il funzionamento dei nodi, e si hanno fatto sempre sopra essempi di nanostack.

Questi essempi compressi nel CD del DevKit, sono piccoli aplicazioni, e sapra tutto sono un ponto dove cominzare a programare una applicazione propia.

Per la realizzazione di questi due progrmmi fatti alla Tesi si ha fatto l'elenco di cominzare sopra il programa di essempio *nano_sensor_n71x.c.*
Queste programma mostra la lettura dei dati dei sensori di forma locale, anche la funzionalità ping e la forma di fare un controllo per i pulsatori.

Ho pensato cominzare sopra queste programma perche all'inizio, il primo scopo era fare una conessione tra due nodi e inviare la informazione di temperatura e luce dello sensore remoto all'altro.
Nelle prossime pagine vado a fare una descrizzione del codice del primo programma, programma fatto per la comunicazione tra solo due node, e inviare e recivire informazione del nodo remoto al nodo locale.



4.2.1 - Primo programa. Invio e Recezione dei dati dei sensori tra due Node.

Como gia si ha detto prima, l'scopo principale di queste programma è la d'ottenere i dati remoti di temperatura e luce del sensore remoto e mostrare questi dati per la consola.
Anche permettere fare l'studio del rango del radio dei sensori a traverso della programmazione della funzionalità di potere cambiare la potenza di trasmisione.

In queste caso, per leggere il port USB si ha fatto l'elenco di utilizare il programa HyperTerminal BBS.

Sono due programmi, Il programma emisore e il programma recetore:

Il codice emisore si programma sul nodo centrale, quello che cominza la trasmisione e il che mostra i dati per consola essendo colegato dal devboard.
Questo programma prevede le seguenti operazioni:

1) Menu di consola per facilitare l'invio di istruzioni a tempo reale dello usuario al nodo centrale.

2) Configurazione locale e del nodo remoto del power di trasmissione, per facilitare l'studio del radio del nodo.

3) Inizio del loop con un tempo introduzido per il usuario che requesta il nodo remoto ogni periodo di tempo e mostra la sua temperatura e luce.

4) Mostra per consola un contatore di petizione inviate al nodo remoto, e un altro di risposti arrivati.

5) Invio di ping, TCP e UDP.

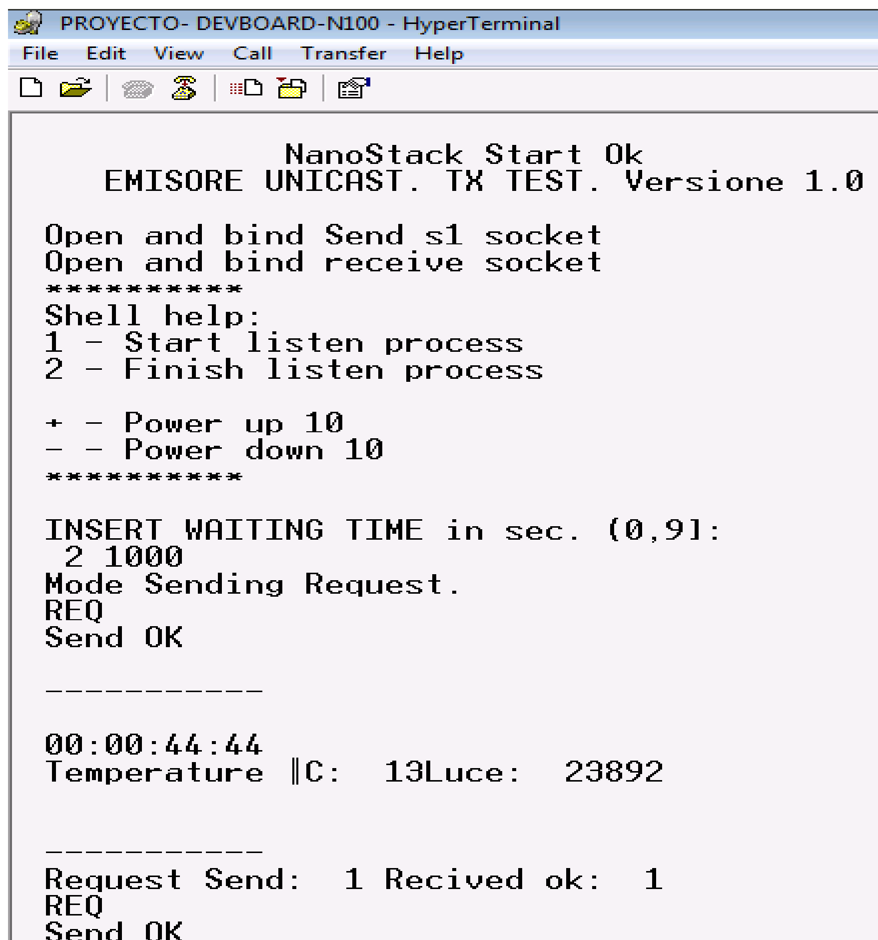
Il codice recettore si programma sul nodo remoto, a fine di inviare sempre che reciva un “request” un packet con la sua temperatura e luce.
Questo programma prevede le seguenti operazioni:

1) Menu di consola per facilitare l'invio di istruzioni a tempo reale dello usuario al nodo.

2) Configurazione locale del power di trasmissione.

3) Invio di ping, TCP e UDP.

La image di sotto mostra l'invio e la recezione d'un pachetto e la impressione dei dati remoti per console, leggendo sempre del nodo emisore o centrale.



4.2.1.1 - Pseudo codice Programma1 – emisore.

```
- Iniziazioni
- Creazione Socket

- for (;;){

    -Sleep 1sec.
    -Lettura del UART 10ms
    -Si arriva qualcosa{

        -arriva 'x'{
            - inizio processo di invio del power al nodo remoto.
        }
        -arriva 'h'{
            -Mostro menu di consola.
        }
        -arriva '1'{
            -Inizio il processo di petizione di temperatura.
            -Petizione all'usuario d'un tempo di retardo tra i petizioni.
        }
        -arriva '+'{
            -Aumento il power +25%
        }
        -arriva '-'{
            -diminuzione del power -25%
        }
        -arriva 'm'{
            -mostro la direzione MAC
        }
        -arriva 'p'{
            -inizio il processo PING
        }
        -arriva 'u'{
            -inizio echo UDP
        }
        -se non arriva nessuno di questi{
            -imprimo il byte che arriva del UART
        }
```

```
-Se doviamo aspetare RISPONSE (activa_temp == 1){

        -mentre non finisca il tempo di recezione{

                -Se arriva in 500ms qualcosa al  socket di recezione{

                        -Se arriva messaggio risponse{
                                -aumento contatore di risponsi arrivatti
                                -mostro per consola la direzzione del packet che arriva.
                                -Prendo i dati di luce e temperatura.
                                -Conversione a Celsius del dato di temperatura.
                                -mostro per consola Temperature e Luce remota.
                        }
                }

                -Se finisce il temporizatore( tempo introdutto per l'usuario ){
                        -inizio variabili per ricomenzare processo di invio di request.
                }
        }

        -Mostro per consola i contatori di packet inviati, packet recivuti
}

-Aspetto 10ms se arriva un packet
-Se arriva il packet{

        -Se è un packet Risponse{
                -Attivo il flag 'activa_temp' .

        }
}



-Se devo fare il invio d'un packet sia request sia controllo di power remoto.{

        -Creazione della variabile dove mettere il packet per fare l'invio.

        -Se la creazione è corretta{
                -Se devo fare l'invio di un messaggio di configurazione del power{
                        -Costruzione del packet.
                }
```

```
                -Se devo fare l'invio di un REQUEST{
                        -Costruzione del packet.
                        -Attivo la variabile di controllo 'attiva_temp'.
                }

                -Invio del packet alla direzione broadcast.
        }
}


-Se il tempo di 'ping' è maggiore di 1sec{
        -Imprimo nello screen i risultati del Ping.
}


Codize del controllo degli "Stack Events" {
        //Codize specifficato negli essempi di nanostack.
}

}
```

4.2.1.2 - Pseudo codice Programma1 – recettore.

```
- Iniziazioni
        - Creazione Socket

        - for (;;){

                -Sleep 1sec.

                -Lettura del UART 10ms
                -Se arriva qualcosa{

                        -arriva '?'{
                                - Mostro il power attuale..
                        }

                        -arriva 'h'{
                                -Mostro menu di consola.
                        }


                        -arriva '+'{
                                -Aumento il power +25%
                        }

                        -arriva '-'{
                                -diminuzione del power -25%
                        }

                        -arriva 'm'{
                                -mostro la direzione MAC
                        }

                        -arriva 'p'{
                                -inizio il processo PING
                        }

                        -arriva 'u'{
                                -inizio echo UDP
                        }
                        -se non arriva nessuno di questi{
                                -imprimo il byte che arriva del UART
                        }
```

```
        -Se arriva un'altra cosa{
                - L'imprimo per consola.
        }
}

- Se ancora non debo inviare  informazione (invio_risponse == 0){
        -Guardo se arriva qualcosa nel buffer di lettura{

                -Se arriva REQUEST{
                        -invio_rispose = 1
                }

                -Se arriva CONF{
                        -Leggo il dato che arriva nel packet,
                        e configuro il mio power con il dato
                        di arrivo.
                }
        }
}

-Se debo inviare informazione (invio_risponse == 1){

        -Creo un buffer per costruire il messaggio.
        -Scribo nel packet il head → RISPONSE.
        -Scribo nel packet i dati di luce e temperatura.
        -Il flag invio_risponse = 0.

        -Invio del Packet.
}




-Se il tempo di 'ping' è maggiore di 1sec{
        -Imprimo nello screen i risultati del Ping.
}



Codize del controllo degli "Stack Events" {
        //Codize specifficato negli essempi di nanostack.
}

}
```

4.2.1 - Secondo programa. Invio e Recezione dei dati dei sensori tra piu di due Node in una topologia di Stella.

In queste caso, l'scoppo principale del programa è la realizzazione di una topologia piu complessa che una sola conessione UNICAST, che è il caso del programa 1 prima spiegatto.

Per la realizzazione d'una soluzione, e dovutto a la poca informazione ottenuta per realizare codice in queste senso, si otta per l'utilizzazione d'una topologia di Stella.

Cosi si pretende realizare un programa capace di racolettare i dati dei nodi remoti in un nodo centrale o emisore, per potere visualizzare nel computer a tempo reale i dati della recezione a traverso d'un lettore del port USB. Il programa scelto, come nel caso anteriore è Hyperterminal.

Piu specifficamente, queste programa funziona cosi:

-Tutti i nodi sono aspettando senza fare niente.

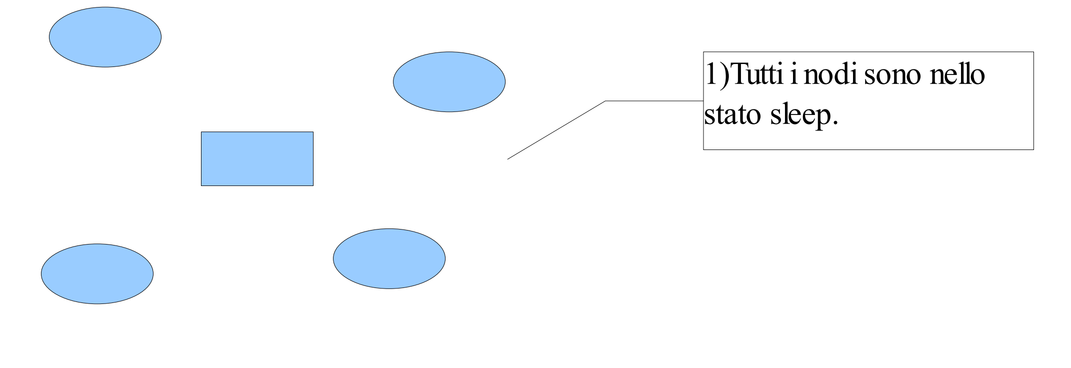
-Cuando il usuario fa la riquesta al nodo centrale di recivire dati dei altri nodi, queste risponde con la petizione di tra cuanto tempo vuole il usuario recivere dati dei nodi.

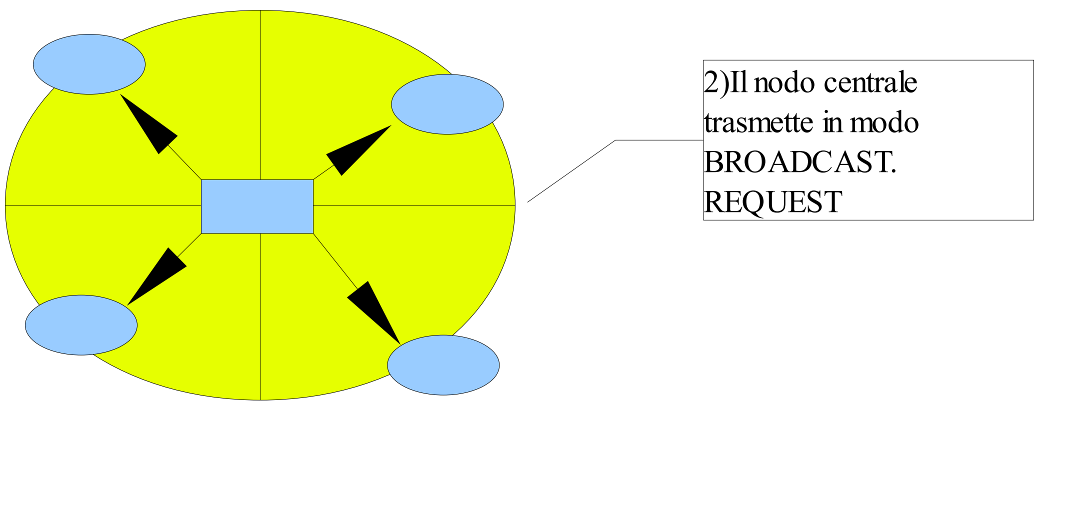
-Il usuario introduze il dato e il programa invia, ogni volta che finisce il tempo di spera introdotto per il usuario, a tutti l'altri nodi un messaggio di "Request".

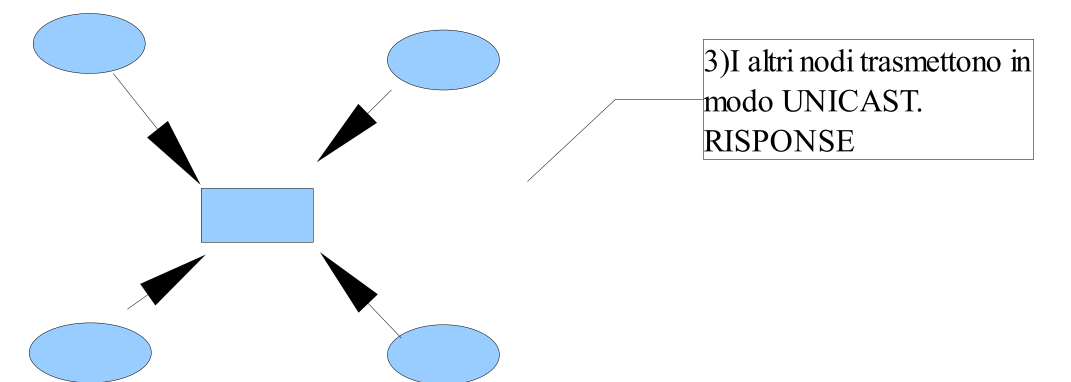
-Gli altri nodi cuando recivono il request fanno l'invio dei dati di luce e temperatura al nodo centrale.

-Il nodo centrale fa la recezione e la visualizzazione di questi dati nello "screen".

Per capire meglio il funzionamento del programa andiamo a mostrare un picolo essempio d'una trasmissione a traverso un schema:







Il codice emisore si programma sul nodo centrale, quello che cominza la trasmisione e il che mostra i dati per consola essendo colegato dal devboard.
Questo programma prevede pratticamente l'stesso che il emisore del programa1, ma con la funzionalità di offrere la recezione di piu nodi, e anche gestire tutta questa informazione per visualizarla nello screen.

Il codice recettore si programma sul nodo remoto, a l'scopo di inviare, sempre che reciva un “request”, un packet con la sua temperatura e luce. Queste programa ha la particolarità di essere diverso in ogni nodo, perche si programa per ogni node l'indirizzo proprio e anche un tempo proprio di retardo nel invio del packets per non produrre colisioni.
Questo programma prevede l'stessa funzionalità che il programa1.

4.2.1.2 - Pseudo codice Programma2 – emisore.

```
- Iniziazioni
        - Creazione Socket

        - for (;;){

                -Sleep 1sec.
                -Lettura del UART 10ms
                -Si arriva qualcosa{

                        -arriva 'x'{
                                - inizio processo di invio del power al nodo remoto.
                        }
                        -arriva 'h'{
                                -Mostro menu di consola.
                        }
                        -arriva '1'{
                                -Inizio il processo di petizione di temperatura.
                                -Petizione all'usuario d'un tempo di retardo tra i petizioni.
                        }
                        -arriva '+'{
                                -Aumento il power +25%
                        }
                        -arriva '-'{
                                -diminuzione del power -25%
                        }
                        -arriva 'm'{
                                -mostro la direzione MAC
                        }
                        -arriva 'p'{
                                -inizio il processo PING
                        }
                        -arriva 'u'{
                                -inizio echo UDP
                        }
                        -se non arriva nessuno di questi{
                                -imprimo il byte che arriva del UART
                        }

                }
```

-Guardo se sono aspetando l'arrivo di packets
-Nel caso di si, inizializo il temporizatore.

-Mentre deva aspetare arrivi di packets{

-Se arriva qualcosa al buffer di recezione{
-Se arriva un packet "RISPONSE"{

-Mostro nello screen la direzione di dove arriva il packet.

-Ricostruzione del dato di temperatura.

-Ricostruzione del dato di Luce.

-Visualizzazione di questi due dati nello screen.

-Contatore speciffico per i nodi dello devKit.(Controllo di funzionamento).

- Libero il buffer di recezione.

}

-Libero il buffer di recezione.

}

-Cuando si hanno inviatto 50 packets, **si ritorna al modo sleep**.
(Non fa niente fino a cuando il usuario ritorna a introdurre '1').

-Se finisce il temporizatore{
-**Ritorno a inviare "REQUEST" in modo broadcast** e usco del loop di recezione.

-Vissualizo nello screen i contatori.

}

}

Tutto questo si fa atraverso l'utilizzazione di flags.

```
-Guardo se devo fare qualcun invio
-Nel caso di dovere farlo{

        -Cominzo la configurazione comune del packet.

        -Se devo inviare un messaggio di configurazione del power.{
                -Confuguro il packet CONF
        }
        -Se devo inviare un messaggio REQUEST{
                -Configuro un messaggio REQUEST.
                -Anche attivo il flag per cominzare il loop di aspeta di risponse.
        }
        -Invio il packet ai nodi, atraverso il socket di invio Broadcast.
        -Se l'invio e corretto{
                -incremento il contatore d'invio.
        }
}


-Se il tempo di 'ping' è maggiore di 1sec{
        -Imprimo nello screen i risultati del Ping.
}


Codize del controllo degli “Stack Events” {
        //Codize specifficato negli essempi di nanostack.
}

}
```

4.2.1.2 - Pseudo codice Programma2 – recettore.

```
- Iniziazioni
-Nella inizaizioni si deve sapere che l'indirizzo viene introdotto per il programatore.
- Creazione Socket

- for (;;){

        -Sleep 1sec.

        -Lettura del UART 10ms
        -Se arriva qualcosa{

                -arriva '?'{
                        - Mostro il power attuale..
                }

                -arriva 'h'{
                        -Mostro menu di consola.
                }


                -arriva '+'{
                        -Aumento il power +25%
                }

                -arriva '-'{
                        -diminuzione del power -25%
                }

                -arriva 'm'{
                        -mostro la direzione MAC
                }

                -arriva 'p'{
                        -inizio il processo PING
                }

                -arriva 'u'{
                        -inizio echo UDP
                }
                -se non arriva nessuno di questi{
                        -imprimo il byte che arriva del UART
                }
```

```
                -Se arriva un'altra cosa{
                        - L'imprimo per consola.
                }
        }

        - Se ancora non debo inviare  informazione (invio_risponse == 0){
                -Guardo se arriva qualcosa nel buffer di lettura{

                        -Se arriva REQUEST{
                                -invio_rispose = 1
                        }

                        -Se arriva CONF{
                                -Leggo il dato che arriva nel packet,
                                e configuro il mio power con il dato
                                di arrivo.
                        }
                }
        }

        -Se debo inviare informazione (invio_risponse == 1){

                -Creo un buffer per costruire il messaggio.
                -Scribo nel packet il head → RISPONSE.
                -Scribo nel packet i dati di luce e temperatura.
                -Il flag invio_risponse = 0.

                -Invio del Packet.
        }



        -Se il tempo di 'ping' è maggiore di 1sec{
                -Imprimo nello screen i risultati del Ping.
        }



        Codize del controllo degli "Stack Events" {
                //Codize specifficato negli essempi di nanostack.
        }

}
```

4.2.1.3 - Discussione dei problemi incontrati.

Come si ha detto programa2 si pretende raggiunggere li scopi che prima abbiamo detto.

Ma, nella elavorazione di queste programa abbiamo trovatto un problema che non abbiamo podutto a la fine soluzionare. Dopo realizare diverse prove, si arriva alla conclusione di afrntare il problema d'una forma organizzata.

Approccio al problema.

-Cercare perche tutti I programi fatti ,per la trasmissione tra piu di due nodi, fanno l'stesa risposta: Dopo diverse trasmissioni I nodi “recettori”, quelli che hanno l'scoppo di inviare al nodo centrale le misure di temperatura e anche la misura di luce, lasciano di trasmettere informazione, fino che solo invia uno di loro, tutti li altri accendono il led 2 e dopo non sonno capaci di ritornare a trasmettere.

Planificazione iniziale per cercare la soluzione al problema del programma broadcast.

-Per fare una aprossimazione a capire il problema e dove succede, si ha pensato in quali possono essere l' cause che provocanno una risposta negativa.

E si ha diciso di essaminare questi tre case:

Tre suposizioni di qual poè essere il problema:

1)¿Essistono colisioni?.

2)¿Il nodo recetore fa bene la memorizzazione della direzione del nodo emisore?.

3)¿Processa il nodo emisore o centrale tutti I packets?.

First step:

Prima di cominzare a risolvere questi problemi, vado a fare una simplificazione del codice nella che potere di forma semplice cercare se qualcuna delle suposizione sonno certe.

Il programa, in modo simplificato, va a essere diverso in ogni nodo recettore, va a includere la direzione propia del nodo e anche la sua propia, cosi cuando arriva il packet request del nodo centrale-emisore, non devo ottenere la direzione e fare il Bind al socket d'invio.

¿Colisioni?. La prima raggione che poè provocare colisioni è che tutti I nodi recettore inviano informazione all'steso tempo al nodo centrale o emisore. Cosi doviamo trovare la forma di cambiare questo cercando come fare per che ogni nodo trasmitta in un tempo diverso. (Nodo1: t1 , Nodo2:t2 , Nodo3:t3)

-Per non trovare il problema dei colisioni, e anche come devo fare un programa per ogni nodo per simplificare e cosi capire meglio quello che sucede, vado a programare un ritardo diverso prima dello invio in ogni nodo recettore, cosi , sono sicuro che l'invio lo fanno in tempi diversi e che non possono essistere colisioni.

Dopo la realizazzione , e il test del programa non arrivo a una conclusione, il problema è ancora l'stesso.

Un altro problema, poè essere che se I nodi recettori non ottengono bene la direzzione del nodo emisore o centrale, invianno l'informazzione a la rete in modo broadcast, e cosi si produrrono colisioni nei nodi recettori.

Ma per fare un test a questa suposizione si programa in ogni nodo l' indirizzo del nodo centrale, e si fa il bind per il socket d'invio. Cosi sapiamo che non si poè dare che I nodi recivanno messagi tra loro, perche solo invianno e recivanno informazioni con il nodo emisore o centrale.

Si salva l'indirizzo del nodo emisore o centrale corretamente? La suposizione è che I nodi recettori salvano corretamente la direzione del nodo centrale, e lavorano con questa invece di con l'indirizzo broadcast della configurazione iniziale nel programa broadcast ,prima di salvare l'indirizzo dopo l'arrivo del primo packet.

In questa suposizione poè sucedere:

Che si vado a imprimire la dirizione a quella invia il nodo recettore, trovo che è la direzione broadcast, e cumunque:

-Questo si puo capitare se il nodo centrale non fa bene l'invio dello indirizzo.

-Il nodo recettore non salva l'indirizzo del nodo centrale correttamente.

-Il nodo recettore non salva la direzzione.

-Il nodo salva però non fa bene la lettura del indirizzo nel momento di fare l'invio.

-Tutti questi suposizioni si possono obbiare limitando il programa ,come si ha spiegato prima, per non realizare il collegamento tra il nodo emisore e il nodo recettore dopo l'invio del primo packet. Nel programa se utilizzano direzzioni direttamente programati nel codice di ogni nodo.

Processa il nodo emisore o centrale tutti I packets??. Poè sucedere che in un programa Cliente – Servitore la recezione nel nodo centrale di tutti I packeti delle altri nodi sia troppo in 1 solo Socket. Per questo si può utilizzare un Servitore Multi-Task, utilizando la funzione fork().

-Dopo realizare l'altre prove, arrivo alla conclussione che il problema non è nel nodo centrale, cosi non vado a provare a realizzare l'uso di fork() perche attualmente il nodo centrale con il codice attuale è capace di processare tutti I packeti che arrivono senza problema. Il problemi sono nel recettore.

Conclusione:

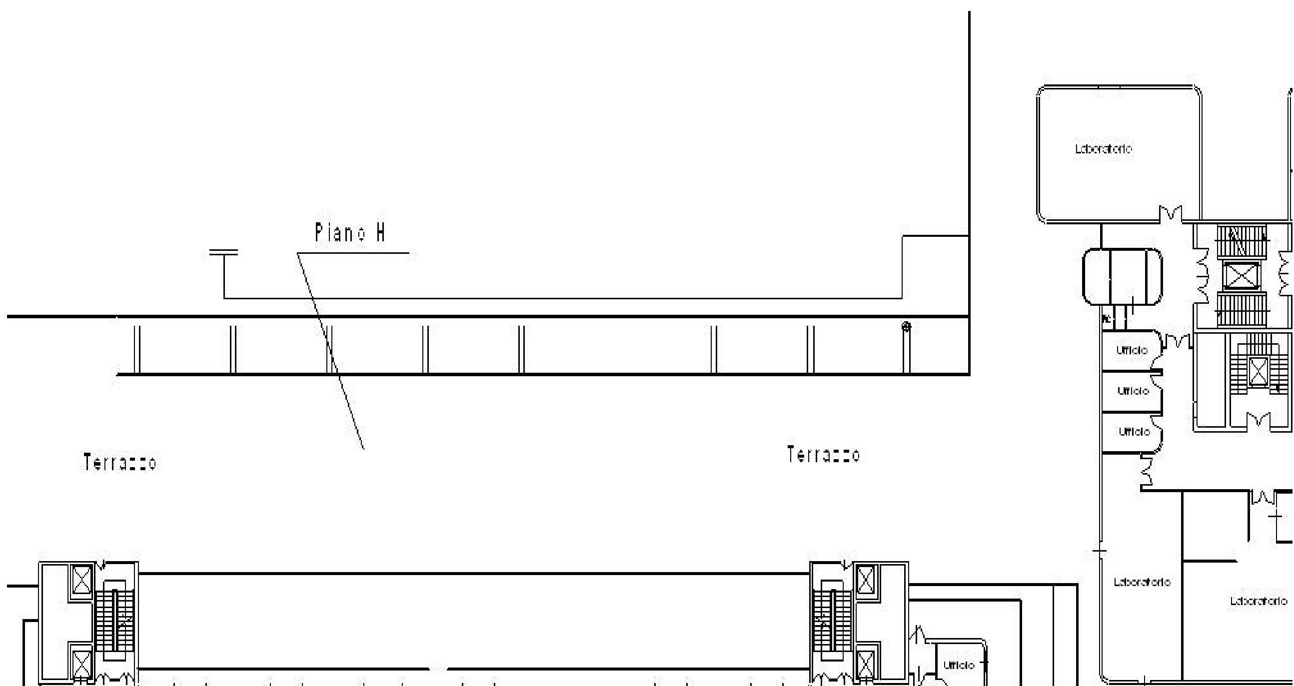
-Il problema sicuramente si trova nella trasmissione , nel programa recettore, o nella cofigurazione dello stack di Sensinode.

-Il programa del nodo centrale funziona correttamente, e anche, quando solo trasmette un nodo recettore, il programa recettore funziona correttamente.

Per qualcuna raggione, e di forma “random” , solo uno dei nodi recettore continua trasmettendo tra un tempo, I altri, accendono il Led2, e poi non fanno la recezione dei dati,. Ma dopo reiniziare questi nodi, succede l'stesso, funzionano bene fino a un momento nel che lascianno di trasmettere.

4.3 - Prove realizate nel laboratorio. Copertura Radio.

Per testare il funzionamento del radio dei sensori si hanno fatto diverse prove di copertura. Soprattuto queste prove sono specifiche per comprobare il comportamento d'una rete WSN d'ultima generazione, sul la tennologia 6LoWPAN, nel speciffico scenario del piano H della Università degli Studi di Pavia. Como centro delle prove si ha utilizzato il laboratorio "Eriksson".



Queste diverse prove, si hanno realizatto con il programma 1 di trasmisione e recezione UNICAST tra solo due nodi.

Ho fatto l'elenco di utilizzare queste programma, perque è piu semplice per testare il Radio Wireless dei nodi, con la intenzione d'elavorare un piccolo test dello funzionamento d'una rete Wireless di sensori con l'stack delle NanoSeries di Sensinode. Sempre sul scenario del laboratorio.

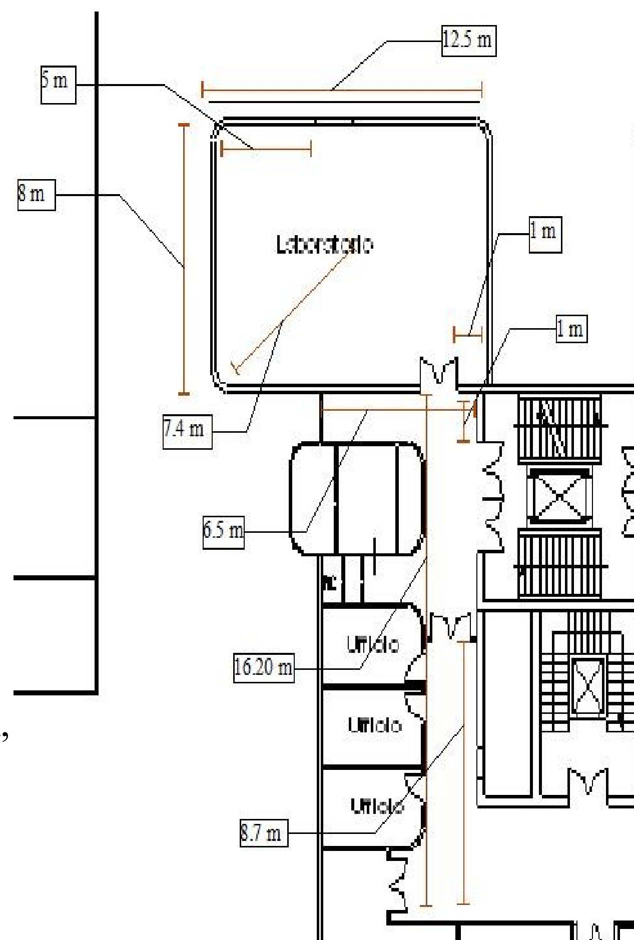
4.3.1 Prima Prova: Comprobare il Radio di Copertura della rete in spazio coperto.

-Como si ha detto gia prima in questa prova si ha utilizzato il programa1, facendo la prova tra due sensori.

-L'scopo di questa prova e fare la stima del radio di copertura d'una rete in spazio coperto, sempre sensa nessuna porta chiusa o finestra tra i due sensori.

-Per realizare la prova si ha fatto un contraste tra il radio di copertura e il power dell'antena del nodo.

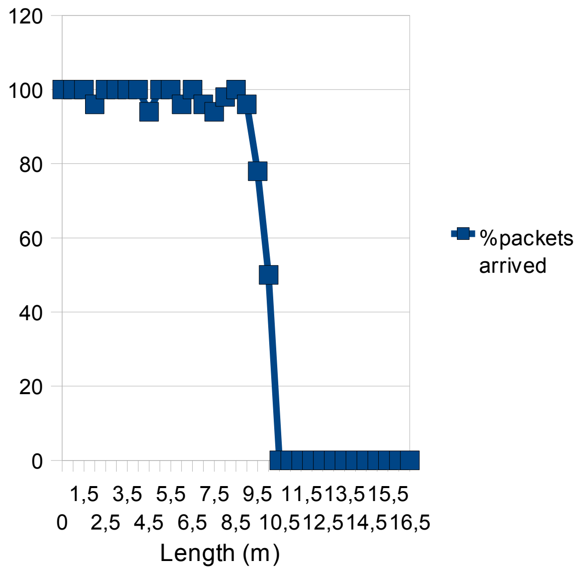
-Si ha realizzato una measurazione como molto da un radio di copertura a tutto lo lungo del corridoio del laboratorio, approssimando sono quasi 16.5 m, sempre contrastando i dati con la potenza. Si ha fatto la measurazione in 25%, 50%, 75% e 100% di potenza.

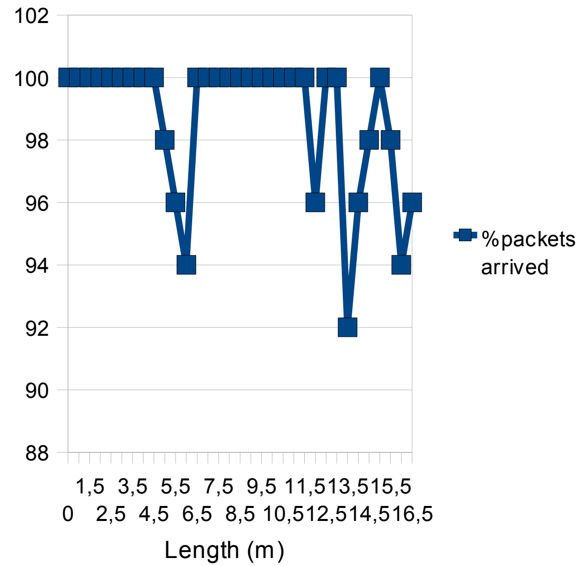


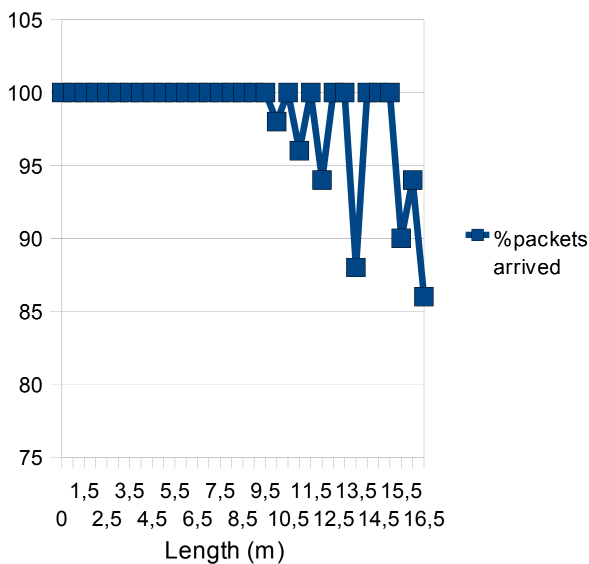
Anche si deve specifficare che nella prova, si invia 1 packet in ogni secondo, e il packet arriva al nodo recettore e dopo , queste risponde. Quando il packet ritorna al nodo emisore è cuando si conta come ok.

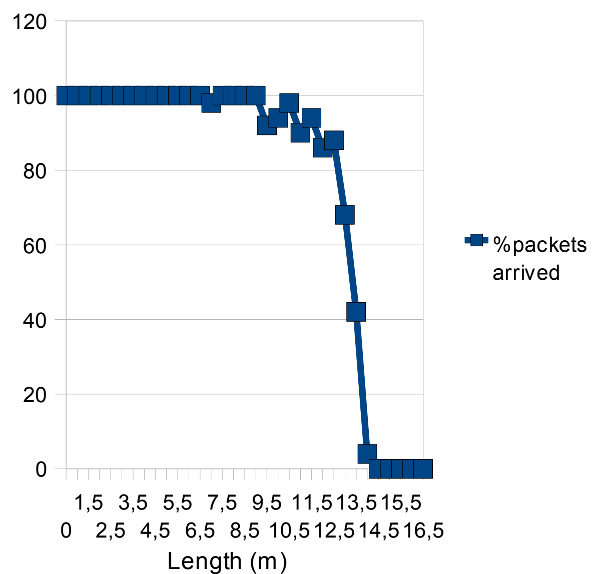
Se il packet è perso nel ritorno o nella andata non si conta come ok.
Queste risultati possono variare un poco secondo il momento della realizzazione della prova, ma come solo vogliamo fare una aprossimazione per sapere piu meno come si deve configurare, secondo il radio, una rete WSN 6LoWPAN al interno del laboratorio, non andiamo a fare più prove come per essempio cercare l'altura ottima per la colocazione del nodo o il angolo nel che si devonno colocare le antene.

Risultati:









Come si può capitare la utilizzazione di piu potenza nella trasmisione si può relazionare direttamente con la distanza del radio di azione del nodo.

Ma qua anche si può capire, che per coprire tutto il corridoio, non è necessaria la utilizzazione del 100% del Power. Perche utilizando un 75% arriva nel caso piu lontano, 16.5 m , piu del 85% dei packets.

Tutto dipende dalla finalità del uso della rete, però in una rete per la measurazione di variabili ambientali, può essere un dato a bastanza buono.

4.3.2 Seconda Prova: *(Experimental Scenarios)* Comprobare il Radio di Copertura della rete nel laboratorio.

Questa prova si forma di diverse prove , piu speciffficamente trè. La finalità di queste prove è comprobare la risposta dei nodi nel Scenario sperimentale del laboratorio.

In queste caso andiamo a fare prove allo interno del laboratorio, prove di copertura del laboratorio al finale del corridoio, e anche un'altra volta nel corridoio per testare a influenza delle porte nella copertura radio dei nodi.

PRIMO TEST: INFLUENZA DELLE PORTE DEL LABORATORIO.

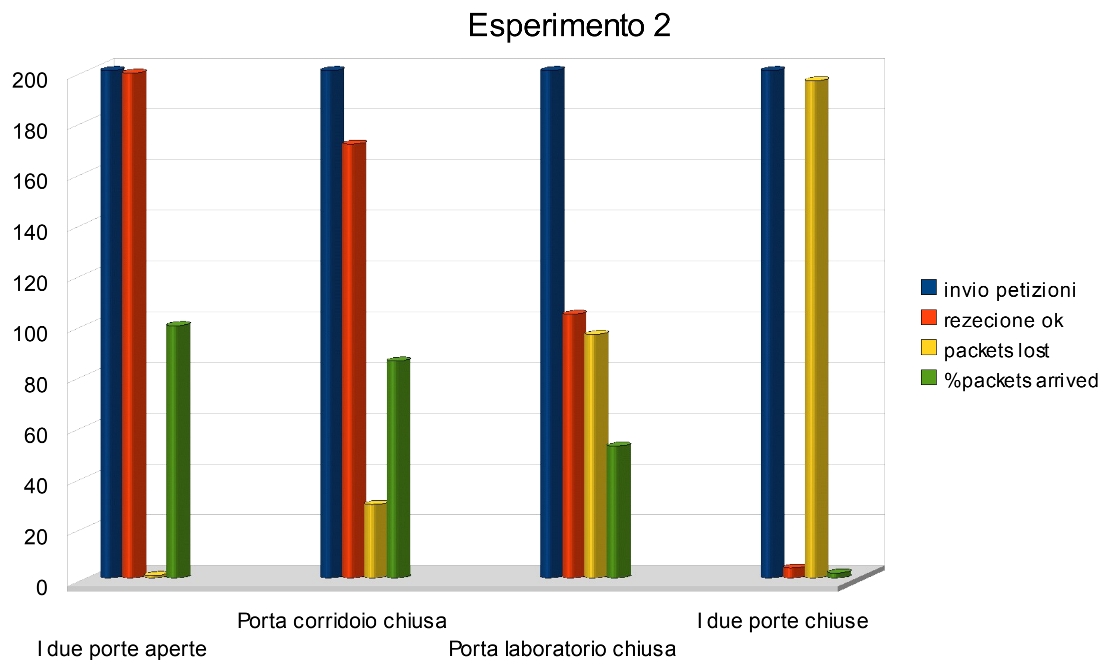
Nel primo test della seconda prova andiamo a testare il funzionamento della trasmisione tra i due nodi facendo distinzioni tra un corridoio completamente aperto o con una o le due porte aperte.

Questa prova è molto speciffica al scenario del laboratorio, gia che non tutte le porte hanno la stessa influenza dipendendo sempre degli materiali utilizatti in questi.

Il power dei nodi si configura al 100%, e le misure si prendono da 4 metri allo interno del laboratorio fino alla fine del corridoio.

Esperimento 2	Descrizione: Da 4 metri in laboratorio fino alla fine del corridoio. POWER = 100%			Length = 23.5 m
	I due porte aperte	Porta corridoio chiusa	Porta laboratorio chiusa	I due porte chiuse
invio petizioni	200	200	200	200
rezecione ok	199	171	104	4
packets lost	1	29	96	196
%packets arrived	99,5	85,5	52	2

Come si può capitare dei risultati, posiamo guardare che dipende dalla lontanezza di dove si trovanno le porte, la influenza nella qualità la temperatura varia. Con le due porte chiuse, pratticamente non c'e comunicazione con tra i nodi , e dipendendo della porta che sia chiusa varia il rango di copertura.
Qua si può guardare cosa succedono d'una forma piu facile di capire:



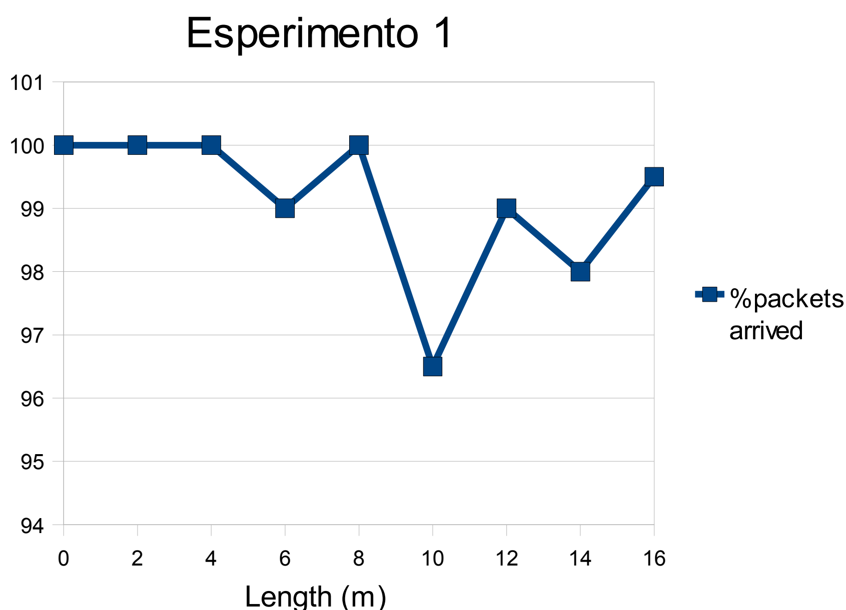
SECONDO TEST: INFLUENZA DELLE PORTE DEL LABORATORIO.

Queste secondo test fa un studio sulla copertura nel corridoio, con la porta di queste chiusa. L' intenzione in queste test piu meno è l'stessa che nel altro realizato su. Ma la differenza è che con questi dati possiamo sapere quanti nodi doviamo spostare per ottenere copertura tra il laboratorio è fino alla fine del corridoio. Sopra tutto, è interessante perche qua utilizzamo un'altra volta la mesaurazione secondo la distanzia.

Esperimento 1	Descrizione: Corridoio del laboratorio con porta chiusa. POWER = 100%						Length = 16.5 m		
	0	2	4	6	8	10	12	14	16
invio petizioni	200	200	200	200	200	200	200	200	200
rezecione ok	200	200	200	198	200	193	198	196	199
packets lost	0	0	0	2	0	7	2	4	1
%packets arrived	100	100	100	99	100	96,5	99	98	99,5

I resultati sono interessanti se ritorniamo a guardare i risultati del primo test. Perche la influenza della porta del corridoio è pratticamente nulla, ma la influenza della porta del laboratorio è a bastanza importante.

Vediamo la comparazione grafica:



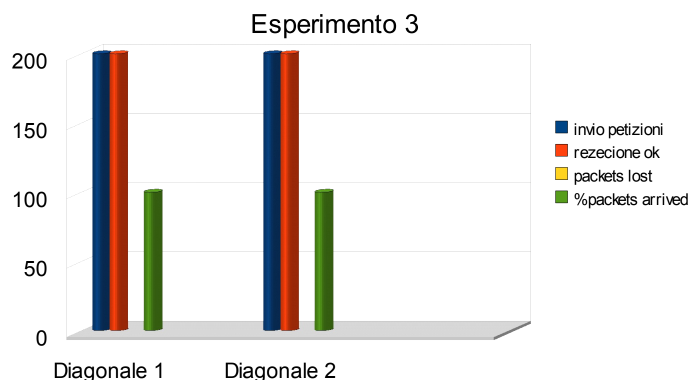
TERZO TEST: COPERTURA NELLE DUE DIAGONALE DEL LABORATORIO.

In queste test andiamo a comprobare che c'e copertura tra tutti le angule del laboratorio, cosi si può capitare che il sistema va a funzionare pinamente senza problemi per la influenza dei computers e tutti le altre elementi che possono produrre disturbi elettromagnetici.

Andiamo a vedere i resultati:

Esperimento 3	Descrizione: Misura degli diagonale del Laboratorio.		Length = 14.8 m
		Diagonale 1	Diagonale 2
	invio petizioni	200	200
	rezecione ok	200	200
	packets lost	0	0
	%packets arrived	100	100

I risultati sono ottimi, si raggiunge il 100% di packets che arrivano correttamente, per tanto si può capitare che non c'e problema alcuno con dove spostare i nodi nel laboratorio.



Capitolo V. Allegati.

Codice programa1. Emisore:

```
/**
 *
 * \file main.c
 * \brief This program test Reception Misures in a WSN Network and provides code to read the sensors on remote
nodes.
 *
 */

/* Standard includes. */
#include <stdlib.h>
#include <string.h>
#include <sys/inttypes.h>


/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* NanoStack includes */
#include "socket.h"
#include "debug.h"
#include "ssi.h"

#include "control_message.h"

/* Platform includes */
#include "uart.h"
#include "rf.h"
#include "bus.h"
#include "dma.h"
#include "timer.h"
#include "gpio.h"
#include "adc.h"

#include "neighbor_routing_table.h"


/* Message types */
#define REQUEST 0x50
#define RISPONSE 0x51
#define CONF 0x52

/*Control Measures*/
#define XTIME 1000 //X * 1000 = X000 Ms
```

```
static void vAppTask( int8_t *pvParameters );

int8_t get_adc_value(adc_input_t channel, uint16_t *value);


ssi_sensor_t ssi_sensor[] =
{/*  ID          | unit type       |scale|data|status*/
  {1, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {2, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {3, SSI_DATA_TYPE_INT,  0,   {0},   0}
};



const uint8_t *ssi_description[] =
{
  "Light",
  "Temp",
        "LEDs"
};

const uint8_t *ssi_unit[] =
{
        "RAW",
        "RAW",
        "xxxxxx21"
};



const uint8_t ssi_n_sensors = sizeof(ssi_sensor)/sizeof(ssi_sensor_t);




/* Setup a default address structure, short address, broadcast, to port 61619 */
sockaddr_t broadcast =
{
        ADDR_BROADCAST,
        { 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        61619
};


sockaddr_t sa  = //Local sensor address
{
        ADDR_802_15_4_PAN_SHORT,
        { 0xFF, 0xFF, 0x3F, 0x12,
        0x02, 0x00, 0x00, 0x20, 0x15, 0x00 },
        61622
};
```

```
xQueueHandle button_events;

/*LED blink times*/
uint16_t led1_count;
uint16_t led2_count;


socket_t *broadcast_socket=0;
socket_t *app_socket;

/* Main task, initialize hardware and start the FreeRTOS scheduler */
int main( void )
{
	/* Initialize the Nano hardware */
	LED_INIT();
	bus_init();
	N710_SENSOR_INIT();


	/* Setup the application task and start the scheduler */
	xTaskCreate( vAppTask, "App", configMINIMAL_STACK_SIZE+200, NULL, (tskIDLE_PRIORITY + 1 ),
( xTaskHandle * )NULL );
	vTaskStartScheduler();

	/* Scheduler has taken control, next vAppTask starts executing. */

	return 0;
}



discover_res_t echo_result;
stack_event_t stack_event;
int8_t ping_active=0;
portTickType ping_start = 0;





/**
 * Application task
 */
static void vAppTask( int8_t *pvParameters )
{



	uint8_t event;
	uint8_t buttons = 0;

	int16_t byte, time, exit;


	buffer_t *buf, *buf_receive, *buf_receivetem;
	uint8_t ind = 0, sending_request = 0, activa_temp = 0, length = 0, header = 0, i= 0,remoto = 0;
```

```
uint16_t rec_start = 0, count_rec = 0,count_env = 0;

uint32_t sum = 0;

uint16_t date_request = 0, luce = 0, temp = 0;
uint16_t U1_value = 0,var1 = 0;
uint16_t U2_value = 0,var2 = 0;
uint8_t count = 0, tx_power = 100;

stack_init_t *stack_rules=0;

uint8_t  primeravez = 0;

pvParameters;



N710_BUTTON_INIT();

/* Start the debug UART at 115k */
debug_init(115200);
button_events = xQueueCreate( 4, 1 /*message size one byte*/ );

led1_count = 50;
led2_count = 100;

vTaskDelay( 50 / portTICK_RATE_MS );
/* Start the debug UART at 115k */
vTaskDelay( 200 / portTICK_RATE_MS );

/* Initialize NanoStack with default parameters, NanoStack task automatically created. */
{
        if(stack_start(NULL)==START_SUCCESS)
        {
                debug("           NanoStack Start Ok\r\n");
                debug("   EMISORE UNICAST. TX TEST. Versione 1.0\r\n\r\n");
        }
}

LED1_ON();
vTaskDelay( 500 / portTICK_RATE_MS );
LED1_OFF();

stack_event                  = open_stack_event_bus();                       /* Open socket for stack status message
*/
stack_service_init( stack_event,NULL, 0 , NULL );  /* No Gateway discover */


//Open and bind Broadcast socket
broadcast_socket = socket(MODULE_CUDP, 0);
if (broadcast_socket) {
        if (socket_bind(broadcast_socket, &broadcast) != pdTRUE)
        {
                debug("Socket bind Send1 failed.\r\n");
        }
        else {
```

```
				debug("Open and bind Send s1 socket\r\n");
		}
	}



	/*Open and bind local socket*/
	app_socket = socket(MODULE_CUDP, 0);
	if (app_socket) {
		if (socket_bind(app_socket, &sa) != pdTRUE)
		{
			debug("Socket bind receive failed.\r\n");
		}
		else {

			debug("Open and bind receive socket\r\n");
		}
	}




	/* Start an endless task loop, we must sleep most of the time allowing execution of other tasks. */
	for (;;)
	{


		/* Sleep for 1000 ms */
		vTaskDelay( 1000 / portTICK_RATE_MS );

/**********************************************************************************************
*******/
	/* Sleep for 10 ms or received from UART */
		byte = debug_read_blocking(10 / portTICK_RATE_MS);
		if (byte != -1)
		{
			switch(byte)
			{

				//Start remote power configuration.
				case 'x':
					if(remoto == 1){

						remoto = 0;
					}else{

							remoto = 1;
							debug("\r\nSending CONF packet");
						}

					break;


				//Shows console
				case 'h':
					debug("********** \r\n");
					debug("Shell help:\r\n1 - Start listen process\r\n2 - Finish listen
```

```
process\r\n");
                    debug("\r\n+ - Power up 10\r\n- - Power down 10");
                    debug("\r\n********** \r\n");

                    break;


               //Start Request process; Sends Request Packets to Remote node. Includes a routine
time entered by the user.
               case '1':

                    if(sending_request == 0){

                         LED1_ON();
                         LED2_ON();


                         time = 0;
                         debug("\r\nINSERT WAITING TIME in sec. (0,9]: \r\n");

                         LED1_ON();
                         LED2_ON();

                         time = debug_read_blocking(10000 / portTICK_RATE_MS);
                         time = time - 48;  //ASCII to decimal.

                         if (time >= 0 && time < 10 ){

                              debug_int(time);

                              time = time * XTIME;
                              time = time - 1000;

                              debug_int(time);
                              LED1_OFF();
                              LED2_OFF();

                              debug("\r\nMode Sending Request.");
                              debug("\r\n");

                              LED1_ON();
                              LED2_ON();

                              sending_request = 1;


                         }else{
                              debug("Error introducted time");
                              time = 0;

                         }

                    break;


               //Local Tx power configuration
               case '+':
```

```
			if(tx_power==100){
				debug("Max Tx power set up.\r\n");

			}else{
					tx_power += 25;
					rf_power_set(tx_power);
					debug("Current power ");
					debug_int(tx_power);
					debug("\r\n");
				}
				break;

		case '-':
			if(tx_power==25){
				debug("Min Tx power set up 25.\r\n");

			}else{
					tx_power -= 25;
					rf_power_set(tx_power);
					debug("Current power ");
					debug_int(tx_power);
					debug("\r\n");
				}
				break;


		case '\r':
			debug("\r\n");
			break;

		//Prints Local mac
		case 'm':
			{
				sockaddr_t mac;

				rf_mac_get(&mac);

				debug("MAC: ");
				debug_address(&mac);
				debug("\r\n");
			}
			break;


		//Start Ping Process
		case 'p':
			if(ping_active == 0)
			{
				echo_result.count=0;
				if(ping(NULL, &echo_result) == pdTRUE) /* Broadcast */
				{
					ping_start = xTaskGetTickCount();
					ping_active = 2;
					debug("Ping\r\n");
				}
				else
					debug("No buffer.\r\n");
			}
```

```
                                        break;

                                case 'u':
                                        if(ping_active == 0)
                                        {
                                                echo_result.count=0;
                                                if(udp_echo(NULL, &echo_result) == pdTRUE)
                                                {
                                                        ping_start = xTaskGetTickCount();
                                                        ping_active = 1;
                                                        debug("udp echo_req()\r\n");
                                                }
                                                else
                                                        debug("No buffer.\r\n");
                                        }
                                        break;

                                default:
                                        debug_put(byte);
                                        break;
                        }
                }

                //Time and recived loop for Sending Recived process
                        if(activa_temp == 1){

                                rec_start = xTaskGetTickCount();

                                debug("\r\n-----------\r\n");

                                while(activa_temp == 1){

                                        //Loop for Reading packets
                                        buf_receivetem = socket_read(app_socket, 500);
                                        if(buf_receivetem){        //Arriva il messagge.
                                                ind = 0;
                                                ind = buf_receivetem->buf_ptr;
                                                length = buf_receivetem->buf_end - buf_receivetem->buf_ptr;
                                                header = buf_receivetem->buf[ind++];

                                                if(header == RISPONSE){
                                                        count_rec ++;
                                                        debug("\r\n");


                                                        for(i=0;i<4;i++){
                                                                debug_hex(buf_receivetem->src_sa.address[9-
i]);
                                                                if(i!=3){
                                                                        debug(":");
                                                                }
                                                        }

                                                        //Reconstruction of Remote Light data
                                                        var1 = buf_receivetem->buf[ind++];
                                                        var2 = buf_receivetem->buf[ind++];
                                                        var1 = var1 << 8;
                                                        var1 = var1 + var2;
```

```
luce = var1;

//Reconstruction of Remote Temperature data
var1
= buf_receivetem->buf[ind++];
var2 = buf_receivetem->buf[ind++];
var1 = var1 << 8;
var1 = var1 + var2;

temp = var1;

var1 = var2 = 0;

sum = (uint32_t) temp;

//Conversion of temperature data in Celsius
sum *= 122;
sum /= 10000;
sum -= 68;

temp = (uint16_t) sum;



debug("\r\nTemperature ºC: ");
debug_int(temp);
debug("Luce: ");
debug_int(luce);




debug("\r\n\r\n");
stack_buffer_free(buf_receivetem);

        }
    }


    if (((xTaskGetTickCount() - rec_start)*portTICK_RATE_MS) >  time ){


        i = 0;
        sending_request = 1;
        activa_temp = 0;
        LED1_OFF();
        LED2_OFF();
        debug("\r\n-----------\r\n");


    }

}

debug("Request Send: ");
```

```
				debug_int(count_env);
				debug(" Recived ok: ");
				debug_int(count_rec);
				debug("\r\n");
			}




		/////////////////////////////////////
		//	Code for received messages	//
		/////////////////////////////////////


			buf_receive = socket_read(app_socket, 10);

			if(buf_receive){   //Message Arrived

				ind = 0;
				ind = buf_receive->buf_ptr;
				length = buf_receive->buf_end - buf_receive->buf_ptr;
				header = buf_receive->buf[ind++];

				//	Reading the Head of Message
				if(header == RISPONSE){ //Arrived RISPONSE message.

						activa_temp = 1;

				}else{
					debug("\r\nArrived Risponse out of Time\r\n");
				}
				stack_buffer_free(buf_receive);
			}



		/////////////////////////////////////
		//	  Code for Send messages	//
		/////////////////////////////////////

		if((sending_request == 1)||(remoto == 1)){


			//Construction of messages
			buf = socket_buffer_get(broadcast_socket);

			if (buf) {
				buf->buf_end=0;
				buf->buf_ptr=0;
				buf->options.hop_count = 1;// Hop = 1, Star Topology, only one hop.

				//Construction of Power configuration messages
				if(remoto == 1){
					buf->buf[buf->buf_end++] = CONF;
					buf->buf[buf->buf_end++] = tx_power;
					debug("CONF ");
					remoto = 0;
```

```
                //Solo un unico invio di request
            }

            //Construction of REQUEST messages
            if(sending_request == 1){

                debug("REQ");
                buf->buf[buf->buf_end++] = REQUEST;
                activa_temp = 1;

                //Solo un unico invio di request

            }


            if (socket_sendto(broadcast_socket, &broadcast, buf) == pdTRUE) {

                    if(sending_request == 1){
                        count_env++;
                        sending_request = 0;
                    }

                    debug("\r\nSend OK");

                }else{
                    debug("\r\nSEND FAILED\r\n");
                }

        }else{
            debug("\r\nError: socket_buffer_get(). Any buffer created\r\n");
        }
}



/* ping response handling */
if ((xTaskGetTickCount() - ping_start)*portTICK_RATE_MS > 1000 && ping_active)
{
        debug("Ping timeout.\r\n");
        stop_ping();
        if(echo_result.count)
        {
            debug("Response: \r\n");
            for(i=0; i<echo_result.count; i++)
            {
                debug_address(&(echo_result.result[i].src));
                debug(" ");
                debug_int(echo_result.result[i].rssi);
                debug(" dbm, ");
                debug_int(echo_result.result[i].time);
                debug(" ms\r\n");
            }
            echo_result.count=0;

        }
        else
        {
            debug("No response.\r\n");
```

```
				}
				ping_active = 0;
			}

			/* stack events */
			if(stack_event)
			{
				buffer_t *buffer = waiting_stack_event(10);
				if(buffer)
				{
					switch (parse_event_message(buffer))
					{
						case BROKEN_LINK:
							debug("Route broken to ");
							debug("\r\n");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						case NO_ROUTE_TO_DESTINATION:
							debug("ICMP message back, no route ");
							debug("\r\n");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						case TOO_LONG_PACKET:
							debug("Payload Too Length\r\n");
							break;

						case DATA_BACK_NO_ROUTE:
							debug("DATA back, No route");
							debug("\r\n");
							debug("To ");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						default:

							break;
					}
					if(buffer)
					{
						socket_buffer_free(buffer);
						buffer = 0;
					}
				}
			} /*end stack events*/
		} /*end main loop*/
}


int8_t get_adc_value(adc_input_t channel, uint16_t *value)
{
		int8_t retval;
```

```
	if (adc_convert_single(channel, ADCREF_AVDD, ADCRES_14BIT) == 0)
	{
		retval = 0;
		while (retval != 1)
		{
			retval = adc_result_single(value);
		}
		retval = 0;
	}
	else
	{
		retval = -1;
	}

	return retval;
}
```

Codice programa1. Recettore:

```
/**
 *
 * \file main.c
 * \brief This program test Reception Misures in a WSN Network and provides code to read the sensors on remote
nodes.
 *
 */

/* Standard includes. */
#include <stdlib.h>
#include <string.h>
#include <sys/inttypes.h>


/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* NanoStack includes */
#include "socket.h"
#include "debug.h"
#include "ssi.h"

#include "control_message.h"

/* Platform includes */
#include "uart.h"
#include "rf.h"
#include "bus.h"
#include "dma.h"
#include "timer.h"
#include "gpio.h"
#include "adc.h"

#include "neighbor_routing_table.h"


/* Message types */
#define REQUEST 0x50
#define RISPONSE 0x51
#define CONF 0x52

/*Control Measures*/
#define XTIME 1000 //X * 1000 = X000 Ms


static void vAppTask( int8_t *pvParameters );

int8_t get_adc_value(adc_input_t channel, uint16_t *value);
```

```
ssi_sensor_t ssi_sensor[] =
{/*  ID           | unit type        |scale|data|status*/
  {1, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {2, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {3, SSI_DATA_TYPE_INT,  0,   {0},   0}
};



const uint8_t *ssi_description[] =
{
  "Light",
  "Temp",
        "LEDs"
};

const uint8_t *ssi_unit[] =
{
        "RAW",
        "RAW",
        "xxxxxx21"
};




const uint8_t ssi_n_sensors = sizeof(ssi_sensor)/sizeof(ssi_sensor_t);





/*Setup a default address structure, short address, to port 61622 */


sockaddr_t dirsens =
{
        ADDR_802_15_4_PAN_SHORT,
        { 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        61622
};


sockaddr_t sa  =
{
        ADDR_802_15_4_PAN_SHORT,
        { 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        61619
};

xQueueHandle button_events;

/*LED blink times*/
uint16_t led1_count;
```

```
uint16_t led2_count;


socket_t *broadcast_socket=0;

socket_t *app_socket;

/* Main task, initialize hardware and start the FreeRTOS scheduler */
int main( void )
{
        /* Initialize the Nano hardware */
        LED_INIT();
        bus_init();
        N710_SENSOR_INIT();


        /* Setup the application task and start the scheduler */
        xTaskCreate( vAppTask, "App", configMINIMAL_STACK_SIZE+200, NULL, (tskIDLE_PRIORITY + 1 ),
( xTaskHandle * )NULL );

        vTaskStartScheduler();
        /* Scheduler has taken control, next vAppTask starts executing. */

        return 0;
}



discover_res_t echo_result;
stack_event_t stack_event;
int8_t ping_active=0;
portTickType ping_start = 0;






/**
 * Application task
 */
static void vAppTask( int8_t *pvParameters )
{

        uint8_t event;
        uint8_t buttons = 0;
        uint8_t s1_count = 0;
        uint8_t s2_count = 0;
        int16_t byte, time;
        uint8_t i =0,pos = 0;
        uint8_t channel;
        buffer_t *buf, *buf_receive;
        uint8_t length = 0, tx_power = 100;

        uint16_t date_request = 0;
        uint16_t U1_value = 0,var1 = 0;
        uint16_t U2_value = 0,var2 = 0;
        uint8_t count = 0;
```

```
	stack_init_t *stack_rules=0;

	uint8_t ind=0, header=0,sending_request=0,invio_risponse=0, activa_temp = 0, option = 0,primeravez = 0;
	uint16_t rec_start=0;


	pvParameters;

	N710_BUTTON_INIT();

	/* Start the debug UART at 115k */
	debug_init(115200);
	button_events = xQueueCreate( 4, 1 /*message size one byte*/ );

	led1_count = 50;
	led2_count = 100;

	vTaskDelay( 50 / portTICK_RATE_MS );
	/* Start the debug UART at 115k */
	vTaskDelay( 200 / portTICK_RATE_MS );


	/* Initialize NanoStack with default parameters, NanoStack task automatically created. */
	{
		if(stack_start(NULL)==START_SUCCESS)
		{
			debug("		NanoStack Start Ok\r\n");
			debug("	RECEPTORE UNICAST. TX TEST. Versione 1.0\r\n\r\n");
		}
	}

	LED1_ON();
	vTaskDelay( 500 / portTICK_RATE_MS );
	LED1_OFF();

	stack_event			= open_stack_event_bus();			/* Open socket for stack status message
*/
	stack_service_init( stack_event,NULL, 0 , NULL );	/* No Gateway discover */


	/* Open and bind a socket send UNICAST*/
	broadcast_socket = socket(MODULE_CUDP, 0);
	if (broadcast_socket) {
		if (socket_bind(broadcast_socket, &dirsens) != pdTRUE)
		{
			debug("Socket bind Send failed.\r\n");
		}
		else {

			debug("Open and bind Send socket\r\n");
		}
	}

	/* Open and bind a socket receive Port 61620 */
	app_socket = socket(MODULE_CUDP, 0);
	if (app_socket) {
		if (socket_bind(app_socket, &sa) != pdTRUE)
```

```
                {
                        debug("Socket bind receive failed.\r\n");
                }
                else {

                        debug("Open and bind receive socket\r\n");
                }
        }


        /* Start an endless task loop, we must sleep most of the time allowing execution of other tasks. */
        for (;;)
        {


                /* Sleep for 1000 ms */
                vTaskDelay( 1000 / portTICK_RATE_MS );

/***************************************************************************************************
*******/
        /* Sleep for 10 ms or received from UART */
                byte = debug_read_blocking(10 / portTICK_RATE_MS);
                if (byte != -1)
                {
                        switch(byte)
                        {

                                case 'h':
                                        debug("********** \r\n");
                                        debug("Shell help:\r\n");
                                        debug("\r\n+ - Power up 10\r\n- - Power down 10");
                                        debug("\r\n********** \r\n");

                                        break;

                                case '?':
                                        debug("\r\nCurrent power: ")
                                        debug_int(tx_power);
                                        debug("\r\n");
                                        break;

                                case '\r':
                                        debug("\r\n");
                                        break;

                                case '+':
                                        if(tx_power==100){
                                                debug("Max Tx power set up.\r\n");

                                        }else{
                                                        tx_power += 25;
                                                        rf_power_set(tx_power);
                                                        debug("Current power ");
                                                        debug_int(tx_power);
                                                        debug("\r\n");
                                                }
                                                break;
```

```
case '-':
	if(tx_power==25){
		debug("Min Tx power set up 10.\r\n");

	}else{
			tx_power -= 25;
			rf_power_set(tx_power);
			debug("Current power ");
			debug_int(tx_power);
			debug("\r\n");
		}
		break;

case 'm':
	{
		sockaddr_t mac;

		rf_mac_get(&mac);

		debug("MAC: ");
		debug_address(&mac);
		debug("\r\n");
	}
	break;

case 'p':
	if(ping_active == 0)
	{
		echo_result.count=0;
		if(ping(NULL, &echo_result) == pdTRUE) /* Broadcast */
		{
			ping_start = xTaskGetTickCount();
			ping_active = 2;
			debug("Ping\r\n");
		}
		else
			debug("No buffer.\r\n");
	}
	break;

case 'u':
	if(ping_active == 0)
	{
		echo_result.count=0;
		if(udp_echo(NULL, &echo_result) == pdTRUE)
		{
			ping_start = xTaskGetTickCount();
			ping_active = 1;
			debug("udp echo_req()\r\n");
		}
		else
			debug("No buffer.\r\n");
	}
	break;

default:
	debug_put(byte);
	break;
```

```
		}
	}


	////////////////////////////////////
	//		Code for received messages	//
	////////////////////////////////////

	if(invio_risponse == 0){

		buf_receive = socket_read(app_socket, 100);

		if(buf_receive){	//Message arrived


			ind = 0;
			ind = buf_receive->buf_ptr;
			length = buf_receive->buf_end - buf_receive->buf_ptr;
			header = buf_receive->buf[ind++];

			//Read Messagge Head
			if(header == REQUEST){

				//If "invio_risponse == 1" can not receive messages Request
				if(invio_risponse == 0){



					debug("\r\nREQUEST ARRIVED");
					invio_risponse = 1;
					debug("\r\nMode Sending Measure");


				}else{
					debug("\r\nRefused REQUEST PACKET\r\n");
				}
			}else{
				debug("\r\nNo Req mesagge\r\n");

			}

			if(header == CONF){
				tx_power = buf_receive->buf[ind++];
				rf_power_set(tx_power);

			}


			stack_buffer_free(buf_receive);
		}

	}else{
		debug("\r\nNo entro a recibir\r\n");
	}
```

```
//////////////////////////////////
//Codize dell'invio dei messaggi.//
//////////////////////////////////

if(invio_risponse == 1){

        //Construction of messages
        buf = socket_buffer_get(broadcast_socket);

        if (buf) {
                buf->buf_end=0;
                buf->buf_ptr=0;
                buf->options.hop_count = 1;// Hop = 1, perche in queste essempio è la massima
distanza posibile.

                //Construction of packet RISPONSE
                //Read Sensor data.
                if(get_adc_value(N710_LIGHT, &U1_value) == 0){

                        if(get_adc_value(N710_TEMP, &U2_value) == 0){

                                debug("\r\nCreating Packet risponse");
                                buf->buf[buf->buf_end++] = RISPONSE;


                                buf->buf[buf->buf_end++] = (U1_value >> 8);
                                buf->buf[buf->buf_end++] = (uint8_t) U1_value;

                                buf->buf[buf->buf_end++] = (U2_value >> 8);
                                buf->buf[buf->buf_end++] = (uint8_t) U2_value;

                                invio_risponse = 0;

                                }else{
                                        debug("\r\nError: Failed Read Sensor Measure\r\n");
                                }


                        }else{
                                debug("\r\nError: Failed Read Sensor Measure\r\n");
                        }




                if (socket_sendto(broadcast_socket, &dirsens, buf) == pdTRUE) {
                        debug("\r\nSend OK -1sec Sleep-");




                }else{
                        debug("\r\nSEND FAILED\r\n");
                }
```

```
			}else{
				debug("\r\nError: socket_buffer_get(). Any buffer created\r\n");
			}
		}


		/* ping response handling */
		if ((xTaskGetTickCount() - ping_start)*portTICK_RATE_MS > 1000 && ping_active)
		{
			debug("Ping timeout.\r\n");
			stop_ping();
			if(echo_result.count)
			{
				debug("Response: \r\n");
				for(i=0; i<echo_result.count; i++)
				{
					debug_address(&(echo_result.result[i].src));
					debug(" ");
					debug_int(echo_result.result[i].rssi);
					debug(" dbm, ");
					debug_int(echo_result.result[i].time);
					debug(" ms\r\n");
				}
				echo_result.count=0;
			}
			else
			{
				debug("No response.\r\n");
			}
			ping_active = 0;
		}

		/* stack events */
		if(stack_event)
		{
			buffer_t *buffer = waiting_stack_event(10);
			if(buffer)
			{
				switch (parse_event_message(buffer))
				{
					case BROKEN_LINK:
						debug("Route broken to ");
						debug("\r\n");
						debug_address(&(buffer->dst_sa));
						debug("\r\n");
						break;

					case NO_ROUTE_TO_DESTINATION:
						debug("ICMP message back, no
route ");
						debug("\r\n");
						debug_address(&(buffer->dst_sa));
						debug("\r\n");
						break;

					case TOO_LONG_PACKET:
						debug("Payload Too Length\r\n");
```

```
							break;

						case DATA_BACK_NO_ROUTE:
							debug("DATA back, No route");
							debug("\r\n");
							debug("To ");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						default:

							break;
					}
					if(buffer)
					{
						socket_buffer_free(buffer);
						buffer = 0;
					}
				}
			} /*end stack events*/
		} /*end main loop*/
}


int8_t get_adc_value(adc_input_t channel, uint16_t *value)
{
	int8_t retval;

	if (adc_convert_single(channel, ADCREF_AVDD, ADCRES_14BIT) == 0)
	{
		retval = 0;
		while (retval != 1)
		{
			retval = adc_result_single(value);
		}
		retval = 0;
	}
	else
	{
		retval = -1;
	}

	return retval;
}
```

Codice Programa2 – Emisore.

```
/**
 *
 * \file main.c
 * \brief Example bare skeleton for starting a new Nano series application.
 *
 */

/* Standard includes. */
#include <stdlib.h>
#include <string.h>
#include <sys/inttypes.h>


/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* NanoStack includes */
#include "socket.h"
#include "debug.h"
#include "ssi.h"

#include "control_message.h"

/* Platform includes */
#include "uart.h"
#include "rf.h"
#include "bus.h"
#include "dma.h"
#include "timer.h"
#include "gpio.h"
#include "adc.h"

#include "neighbor_routing_table.h"


/* Message types */
#define REQUEST 0x50
#define RISPONSE 0x51
#define CONF 0x52


/*Control Measures*/
#define XTIME 1000 //X * 1000 = X000 Ms


static void vAppTask( int8_t *pvParameters );

int8_t get_adc_value(adc_input_t channel, uint16_t *value);
```

```
ssi_sensor_t ssi_sensor[] =
{/*  ID            | unit type        |scale|data|status*/
  {1, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {2, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {3, SSI_DATA_TYPE_INT,  0,   {0},   0}
};



const uint8_t *ssi_description[] =
{
  "Light",
  "Temp",
        "LEDs"
};

const uint8_t *ssi_unit[] =
{
        "RAW",
        "RAW",
        "xxxxxx21"
};




const uint8_t ssi_n_sensors = sizeof(ssi_sensor)/sizeof(ssi_sensor_t);





/* Setup a default address structure, short address, broadcast, to port 61619 */
sockaddr_t broadcast =
{
        ADDR_BROADCAST,
        { 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
        61619
};


sockaddr_t sa  = //direccion local del sensor
{
        ADDR_802_15_4_PAN_SHORT,
        { 0xFF, 0xFF, 0x7B, 0x10,
        0x02, 0x00, 0x00, 0x20, 0x15, 0x00 },
        61622
};


xQueueHandle button_events;

/*LED blink times*/
uint16_t led1_count;
uint16_t led2_count;
```

```
socket_t *broadcast_socket=0;
socket_t *app_socket;

/* Main task, initialize hardware and start the FreeRTOS scheduler */
int main( void )
{
	/* Initialize the Nano hardware */
	LED_INIT();
	bus_init();
	N710_SENSOR_INIT();


	/* Setup the application task and start the scheduler */
	xTaskCreate( vAppTask, "App", configMINIMAL_STACK_SIZE+200, NULL, (tskIDLE_PRIORITY + 1 ),
( xTaskHandle * )NULL );
	vTaskStartScheduler();

	/* Scheduler has taken control, next vAppTask starts executing. */

	return 0;
}



discover_res_t echo_result;
stack_event_t stack_event;
int8_t ping_active=0;
portTickType ping_start = 0;






/**
 * Skeleton application task
 */
static void vAppTask( int8_t *pvParameters )
{

	uint8_t event;
	uint8_t buttons = 0;
	uint8_t s1_count = 0;
	uint8_t s2_count = 0;
	int16_t byte, time;
	uint8_t i =0,a =0, pos = 0;
	uint8_t channel;
	buffer_t *buf, *buf_receive, *buf_receivetem;
	uint8_t length = 0;


	uint16_t date_request = 0;
	uint16_t U1_value = 0,var1 = 0;
	uint16_t U2_value = 0,var2 = 0,tx_power =100,sum = 0;
	uint8_t count1 = 0, count2= 0, count3= 0, count_env= 0, countpar=0;

	stack_init_t *stack_rules=0;
```

```
	uint8_t ind=0, header=0,sending_request=0,waiting_risponse=0,invio_risponse=0, activa_temp = 0, option = 0,
select_socket=0, remoto = 0;
	uint16_t temp;
	uint16_t luce;
	uint16_t rec_start=0;


	pvParameters;

	N710_BUTTON_INIT();

	/* Start the debug UART at 115k */
	debug_init(115200);
	button_events = xQueueCreate( 4, 1 /*message size one byte*/ );

	led1_count = 50;
	led2_count = 100;

	vTaskDelay( 50 / portTICK_RATE_MS );
	/* Start the debug UART at 115k */
	vTaskDelay( 200 / portTICK_RATE_MS );

	/* Initialize NanoStack with default parameters, NanoStack task automatically created. */
	{
		if(stack_start(NULL)==START_SUCCESS)
		{
			debug("	NanoStack Start Ok\r\n");
			debug("   EMISORE BROADCAST. TX TEST. Versione 1.0\r\n\r\n");
		}
	}

	LED1_ON();
	vTaskDelay( 500 / portTICK_RATE_MS );
	LED1_OFF();

	stack_event			= open_stack_event_bus();			/* Open socket for stack status message
*/
	stack_service_init( stack_event,NULL, 0 , NULL );  /* No Gateway discover */

	channel = mac_current_channel();



	//Open and bind a socket1 send Broadcast
	broadcast_socket = socket(MODULE_CUDP, 0);
	if (broadcast_socket) {
		if (socket_bind(broadcast_socket, &broadcast) != pdTRUE)
		{
			debug("Socket bind Send1 failed.\r\n");
		}
		else {

			debug("Open and bind Send s1 socket\r\n");
		}
	}
```

```
	/*Open and bind a socket app*/
	app_socket = socket(MODULE_CUDP, 0);
	if (app_socket) {
		if (socket_bind(app_socket, &sa) != pdTRUE)
		{
			debug("Socket bind receive failed.\r\n");
		}
		else {

			debug("Open and bind receive socket\r\n");
		}
	}




	/* Start an endless task loop, we must sleep most of the time allowing execution of other tasks. */
	for (;;)
	{


		/* Sleep for 1000 ms */
		vTaskDelay( 100 / portTICK_RATE_MS );

/*************************************************************************************************
*******/
	/* Sleep for 10 ms or received from UART */
		byte = debug_read_blocking(10 / portTICK_RATE_MS);
		if (byte != -1)
		{
			switch(byte)
			{

				case 'x':

					if(remoto == 1){

						remoto = 0;
					}else{

							remoto = 1;
							debug("\r\nSending CONF packet");
						}

					break;


				case 'h':
					debug("********** \r\n");
					debug("Shell help:\r\n1 - Start listen process\r\n2 - Finish listen
process\r\n");

					debug("\r\np - Start ping process\r\nu - Start udp echo_req()");
					debug("\r\n********** \r\n");

					break;
```

```
				case '1':

					if(waiting_risponse == 0 && sending_request == 0 && invio_risponse == 
0){

						LED1_ON();
						LED2_ON();


						time = 0;
						debug("\r\nINSERT time in sec. (0,9]: \r\n");

						LED1_ON();
						LED2_ON();

						time = debug_read_blocking(10000 / portTICK_RATE_MS);
						time = time - 48;  //de ASCII a decimal.

						if (time >= 0 && time < 10 ){

							time = time * XTIME;

							sending_request = 1;


						}else{
							debug("Error time value.");
							time = 0;

						}
					}else{
						debug("\r\nTo finish Receive information Mode. push '2'.\r\n");
					}

					break;


				case '+':

					if(tx_power == 100){
						debug("Max Tx power set up.\r\n");

					}else{
							tx_power += 25;
							rf_power_set(tx_power);
							debug("Current power ");
							debug_int(tx_power);
							debug("\r\n");
						}
						break;

				case '-':
					if(tx_power == 25){
						debug("Min Tx power set up 25%.\r\n");

					}else{
							tx_power -= 25;
```

```
                        rf_power_set(tx_power);
                        debug("Current power ");
                        debug_int(tx_power);
                        debug("\r\n");
                }
                break;


        case '\r':
                debug("\r\n");
                break;

        case 'm':
                {
                        sockaddr_t mac;

                        rf_mac_get(&mac);

                        debug("MAC: ");
                        debug_address(&mac);
                        debug("\r\n");
                }
                break;

        case 'p':
                if(ping_active == 0)
                {
                        echo_result.count=0;
                        if(ping(NULL, &echo_result) == pdTRUE) /* Broadcast */
                        {
                                ping_start = xTaskGetTickCount();
                                ping_active = 2;
                                debug("Ping\r\n");
                        }
                        else
                                debug("No buffer.\r\n");
                }
                break;

        case 'u':
                if(ping_active == 0)
                {
                        echo_result.count=0;
                        if(udp_echo(NULL, &echo_result) == pdTRUE)
                        {
                                ping_start = xTaskGetTickCount();
                                ping_active = 1;
                                debug("udp echo_req()\r\n");
                        }
                        else
                                debug("No buffer.\r\n");
                }
                break;

        case 'C':
                if (channel < 26) channel++;
                channel++;
        case 'c':
```

```
                        if (channel > 11) channel--;
                        mac_set_channel(channel);
                        debug("Channel: ");
                        debug_int(channel);
                        debug("\r\n");
                        break;

                default:
                        debug_put(byte);
                        break;
        }
}



////////////////////////////////////
//Codize di recezione dei messaggi//
////////////////////////////////////


if(activa_temp == 1){
        rec_start = xTaskGetTickCount();
}
while(activa_temp == 1){

        buf_receive = socket_read(app_socket, 10);


        if(buf_receive){   //Arriva il messagge.

                ind = 0;
                ind = buf_receive->buf_ptr;
                length = buf_receive->buf_end - buf_receive->buf_ptr;
                header = buf_receive->buf[ind++];

                //Leggendo il HEAD del Messagge
                if(header == RISPONSE){ //Arriva RISPONSE, guardo se arriva nel tempo corretto

                                countpar ++;
                                debug("\r\n");

                                //Print Mac Address
                                for(i=0;i<4;i++){
                                        debug_hex(buf_receive->src_sa.address[9-i]);
                                        if(i!=3){
                                                debug(":");
                                        }
                                }



                                //Ricostruzione del dato di Temperatura.
                                var1 = buf_receive->buf[ind++];
                                var2 = buf_receive->buf[ind++];
                                var1 = var1 << 8;
                                var1 = var1 + var2;

                                luce = var1;
```

```
                                        //Ricostruzione del dato di Luce.
                                        var1 = buf_receive->buf[ind++];
                                        var2 = buf_receive->buf[ind++];
                                        var1 = var1 << 8;
                                        var1 = var1 + var2;

                                        temp = var1;

                                        var1 = var2 = 0;

                                        sum = (uint32_t) temp;

                                        /*Visualizzazione dei dati di temperatura in Celsius.*/
                                        sum *= 122;
                                        sum /= 10000;
                                        sum -= 68;

                                        temp = (uint16_t) sum;



                                        debug("\r\nTemp C: ");
                                        debug_int(temp);
                                        debug("Luce: ");
                                        debug_int(luce);

                                        switch(buf_receive->src_sa.address[6]){

                                                case '0x8E':
                                                        count1++;
                                                        break;

                                                case '0xE9':
                                                        count2++;
                                                        break;

                                                case '0x44':
                                                        count2++;
                                                        break;

                                                default:
                                                        break;
                                        }

                                        debug("\r\n\r\n");
                                        stack_buffer_free(buf_receive);

                        }

                        stack_buffer_free(buf_receive);
                }



                if(count_env == 50){
```

```
                        i = 0;
                        sending_request = 0;
                        activa_temp = 0;
                        count_env=count1=count2=count3=0;
                }



                if
(((xTaskGetTickCount() - rec_start)*portTICK_RATE_MS) >  time ){

                        i = 0;
                        sending_request = 1;
                        activa_temp = 0;
                        LED1_OFF();
                        LED2_OFF();



                        debug("ReqSend:");
                        debug_int(count_env);
                        debug("\r\nArriv 8E:");
                        debug_int(count1);
                        debug(" Arriv E9:");
                        debug_int(count2);
                        debug(" Arriv 44:");
                        debug_int(count3);
                        debug("\r\n");


                }


        }//activa_temp




        ///////////////////////////////////
        //Codize dell'invio dei messaggi.//
        ///////////////////////////////////

        if((sending_request == 1)||(remoto == 1)){ //debo fare l'invio
                //construzione comune ai due packets

                buf = socket_buffer_get(broadcast_socket);


                if (buf) {
                        buf->buf_end=0;
                        buf->buf_ptr=0;
                        buf->options.hop_count = 2;
```

```
                                if(remoto == 1){
                                        buf->buf[buf->buf_end++] = CONF;
                                        buf->buf[buf->buf_end++] = tx_power;
                                        debug("CONF ");
                                        remoto = 0;
                                        //Solo un unico invio di request
                                }


                                if(sending_request == 1){ //construzione del packet REQUEST

                                        debug("REQ");
                                        buf->buf[buf->buf_end++] = REQUEST;
                                        activa_temp = 1;

                                        //Solo un unico invio di request

                                }


                                if (socket_sendto(broadcast_socket, &broadcast, buf) == pdTRUE) {//invio dei
packets.

                                                if(sending_request == 1){
                                                        count_env++;
                                                        sending_request = 0;
                                                }

                                                debug("\r\nSend OK");

                                        }else{
                                                debug("\r\nSEND FAILED\r\n");
                                        }

                        }else{
                                debug("\r\nError: socket_buffer_get(). Any buffer created\r\n");
                        }
                }



                /* ping response handling */
                if ((xTaskGetTickCount() - ping_start)*portTICK_RATE_MS > 1000 && ping_active)
                {
                        debug("Ping timeout.\r\n");
                        stop_ping();
                        if(echo_result.count)
                        {
                                debug("Risponse: \r\n");
                                for(i=0; i<echo_result.count; i++)
                                {
                                        debug_address(&(echo_result.result[i].src));
                                        debug(" ");
                                        debug_int(echo_result.result[i].rssi);
                                        debug(" dbm, ");
                                        debug_int(echo_result.result[i].time);
                                        debug(" ms\r\n");
                                }
```

```
				echo_result.count=0;

			}
			else
			{
				debug("No response.\r\n");
			}
			ping_active = 0;
		}

		/* stack events */
		if(stack_event)
		{
			buffer_t *buffer = waiting_stack_event(10);
			if(buffer)
			{
				switch (parse_event_message(buffer))
				{
					case BROKEN_LINK:
						debug("Route broken to ");
						debug("\r\n");
						debug_address(&(buffer->dst_sa));
						debug("\r\n");
						break;

					case NO_ROUTE_TO_DESTINATION:
						debug("ICMP message back, no route ");
						debug("\r\n");
						debug_address(&(buffer->dst_sa));
						debug("\r\n");
						break;

					case TOO_LONG_PACKET:
						debug("Payload Too Length\r\n");
						break;

					case DATA_BACK_NO_ROUTE:
						debug("DATA back, No route");
						debug("\r\n");
						debug("To ");
						debug_address(&(buffer->dst_sa));
						debug("\r\n");
						break;

					default:

						break;
				}
				if(buffer)
				{
					socket_buffer_free(buffer);
					buffer = 0;
				}
			}
		} /*end stack events*/
	} /*end main loop*/
}
```

```
int8_t get_adc_value(adc_input_t channel, uint16_t *value)
{
	int8_t retval;

	if (adc_convert_single(channel, ADCREF_AVDD, ADCRES_14BIT) == 0)
	{
		retval = 0;
		while (retval != 1)
		{
			retval = adc_result_single(value);
		}
		retval = 0;
	}
	else
	{
		retval = -1;
	}

	return retval;
}
```

Codice Programa2 – Recettore.

```
/**
 *
 * \file main.c
 *
 *
 */

/* Standard includes. */
#include <stdlib.h>
#include <string.h>
#include <sys/inttypes.h>


/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* NanoStack includes */
#include "socket.h"
#include "debug.h"
#include "ssi.h"

#include "control_message.h"

/* Platform includes */
#include "uart.h"
#include "rf.h"
#include "bus.h"
#include "dma.h"
#include "timer.h"
#include "gpio.h"
#include "adc.h"

#include "neighbor_routing_table.h"


/* Message types */
#define REQUEST 0x50
#define RISPONSE 0x51
#define CONF 0x52


/*Control Measures*/
#define XTIME 1000 //X * 1000 = X000 Ms


static void vAppTask( int8_t *pvParameters );
```

```
int8_t get_adc_value(adc_input_t channel, uint16_t *value);


ssi_sensor_t ssi_sensor[] =
{/*  ID            | unit type        |scale|data|status*/
  {1, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {2, SSI_DATA_TYPE_INT,  0,   {0},   0},
  {3, SSI_DATA_TYPE_INT,  0,   {0},   0}
};



const uint8_t *ssi_description[] =
{
  "Light",
  "Temp",
        "LEDs"
};

const uint8_t *ssi_unit[] =
{
        "RAW",
        "RAW",
        "xxxxxx21"
};




const uint8_t ssi_n_sensors = sizeof(ssi_sensor)/sizeof(ssi_sensor_t);






sockaddr_t dirsens =
{
        ADDR_802_15_4_PAN_SHORT,
        { 0xFF, 0xFF, 0x7B, 0x10,
        0x02, 0x00, 0x00, 0x20, 0x15, 0x00 },
        61622
};


sockaddr_t sa  =
{
        ADDR_802_15_4_PAN_SHORT,
        { 0xFF, 0xFF, 0x8E, 0x11,
        0x02, 0x00, 0x00, 0x20, 0x15, 0x00 },
        61619
};

xQueueHandle button_events;

/*LED blink times*/
uint16_t led1_count;
```

```
uint16_t led2_count;


socket_t *broadcast_socket=0;

socket_t *app_socket;

/* Main task, initialize hardware and start the FreeRTOS scheduler */
int main( void )
{
	/* Initialize the Nano hardware */
	LED_INIT();
	bus_init();
	N710_SENSOR_INIT();


	/* Setup the application task and start the scheduler */
	xTaskCreate( vAppTask, "App", configMINIMAL_STACK_SIZE+200, NULL, (tskIDLE_PRIORITY + 1 ),
( xTaskHandle * )NULL );


	vTaskStartScheduler();

	/* Scheduler has taken control, next vAppTask starts executing. */

	return 0;
}



discover_res_t echo_result;
stack_event_t stack_event;
int8_t ping_active=0;
portTickType ping_start = 0;






/**
 * Skeleton application task
 */
static void vAppTask( int8_t *pvParameters )
{

	uint8_t event;
	uint8_t buttons = 0;
	uint8_t s1_count = 0;
	uint8_t s2_count = 0;
	int16_t byte, time;
	uint8_t i =0,pos = 0;
	uint8_t channel;
	buffer_t *buf, *buf_receive;
	uint8_t length = 0;

	uint16_t date_request = 0;
	uint16_t U1_value = 0,var1 = 0;
```

```
	uint16_t U2_value = 0,var2 = 0;
	uint8_t count = 0;

	stack_init_t *stack_rules=0;

	uint8_t ind=0, header=0,sending_request=0,waiting_risponse=0,invio_risponse=0, activa_temp = 0, option =
0,primeravez = 1, tx_power =100;
	uint16_t rec_start=0;


	pvParameters;

	N710_BUTTON_INIT();

	/* Start the debug UART at 115k */
	debug_init(115200);
	button_events = xQueueCreate( 4, 1 /*message size one byte*/ );

	led1_count = 50;
	led2_count = 100;

	vTaskDelay( 50 / portTICK_RATE_MS );
	/* Start the debug UART at 115k */
	vTaskDelay( 200 / portTICK_RATE_MS );


	/* Initialize NanoStack with default parameters, NanoStack task automatically created. */
	{
		if(stack_start(NULL)==START_SUCCESS)
		{
			debug("		NanoStack Start Ok\r\n");
			debug("   RECEPTORE BROADCAST. TX TEST. Versione 1.0\r\n\r\n");
		}
	}

	LED1_ON();
	vTaskDelay( 500 / portTICK_RATE_MS );
	LED1_OFF();

	stack_event			= open_stack_event_bus();			/* Open socket for stack status message
*/
	stack_service_init( stack_event,NULL, 0 , NULL );  /* No Gateway discover */

	channel = mac_current_channel();





	app_socket = socket(MODULE_CUDP, 0);
	if (app_socket) {
		if (socket_bind(app_socket, &sa) != pdTRUE)
		{
			debug("Socket bind receive failed.\r\n");
		}
		else {

			debug("Open and bind receive socket\r\n");
```

```
				}
		}


		broadcast_socket = socket(MODULE_CUDP, 0);
		if (broadcast_socket) {
				if (socket_bind(broadcast_socket, &dirsens) != pdTRUE)
				{
						debug("\r\nSocket bind Send failed.\r\n");
				}
				else {

						debug("\r\nOpen and bind Send socket\r\n");
				}
		}

		/* Start an endless task loop, we must sleep most of the time allowing execution of other tasks. */
		for (;;)
		{


				/* Sleep for 100 ms */
				vTaskDelay( 100 / portTICK_RATE_MS );

/******************************************************************************************
*******/
		/* Sleep for 10 ms or received from UART */
				byte = debug_read_blocking(10 / portTICK_RATE_MS);
				if (byte != -1)
				{
						switch(byte)
						{

								case 'h':
										debug("********** \r\n");
										debug("Shell help:\r\n");
										debug("\r\np - Start ping process\r\nu - Start udp echo_req()");
										debug("\r\n********** \r\n");

										break;

								case '?':
										debug("\r\nCurrent power: ")
										debug_int(tx_power);
										debug("\r\n");
										break;

								case '+':
										if(tx_power==100){
												debug("Max Tx power set up.\r\n");

										}else{
														tx_power += 25;
														rf_power_set(tx_power);
														debug("Current power ");
														debug_int(tx_power);
														debug("\r\n");
												}
```

```
				break;

		case '-':
			if(tx_power==25){
				debug("Min Tx power set up 10.\r\n");

			}else{
					tx_power -= 25;
					rf_power_set(tx_power);
					debug("Current power ");
					debug_int(tx_power);
					debug("\r\n");
				}
				break;

		case '\r':
			debug("\r\n");
			break;

		case 'm':
			{
				sockaddr_t mac;

				rf_mac_get(&mac);

				debug("MAC: ");
				debug_address(&mac);
				debug("\r\n");
			}
			break;

		case 'p':
			if(ping_active == 0)
			{
				echo_result.count=0;
				if(ping(NULL, &echo_result) == pdTRUE) /* Broadcast */
				{
					ping_start = xTaskGetTickCount();
					ping_active = 2;
					debug("Ping\r\n");
				}
				else
					debug("No buffer.\r\n");
			}
			break;

		case 'u':
			if(ping_active == 0)
			{
				echo_result.count=0;
				if(udp_echo(NULL, &echo_result) == pdTRUE)
				{
					ping_start = xTaskGetTickCount();
					ping_active = 1;
					debug("udp echo_req()\r\n");
				}
				else
					debug("No buffer.\r\n");
```

```
                }
                break;

            case 'C':
                if (channel < 26) channel++;
                channel++;
            case 'c':
                if (channel > 11) channel--;
                mac_set_channel(channel);
                debug("Channel: ");
                debug_int(channel);
                debug("\r\n");
                break;

            default:
                debug_put(byte);
                break;
        }
    }


    ////////////////////////////////////
    //Codize di recezione dei messaggi//
    ////////////////////////////////////

    if(invio_risponse == 0){

        buf_receive = socket_read(app_socket, 100);


        //Cuando se desconecta llega justo aqui
        if(buf_receive){  //Arriva il messagge.

            if (buf_receive->dst_sa.port == 61619){     //Comprobation Reception Port first of
Start Recived Process.

                ind = 0;
                ind = buf_receive->buf_ptr;
                length = buf_receive->buf_end - buf_receive->buf_ptr;
                header = buf_receive->buf[ind++];

                //Leggendo il HEAD del Messagge
                if(header == REQUEST){

                    if(waiting_risponse == 0 && invio_risponse == 0){//Solo
con questi condizioi posso recivire REQUEST

                        invio_risponse = 1;


                    }else{
                        debug("\r\nRefused REQUEST PACKET\r\n");
                    }
                }


                if(header == CONF){                 //Packet to Power
Configuration.
```

```
                                        tx_power = buf_receive->buf[ind++];
                                        rf_power_set(tx_power);

                                        LED2_ON();
                                        vTaskDelay( 1000 / portTICK_RATE_MS );
                                        LED2_OFF();

                                }


                        }
                }
                stack_buffer_free(buf_receive);
        }







        ////////////////////////////////////
        //Codize dell'invio dei messaggi.//
        ////////////////////////////////////

        if(invio_risponse == 1){ // debo fare l'invio
                //construzione comune ai due packets
                buf = socket_buffer_get(broadcast_socket);



                if (buf) {
                        buf->buf_end=0;
                        buf->buf_ptr=0;
                        buf->options.hop_count = 1;


                        //construzione del packet RISPONSE

                        if(get_adc_value(N710_LIGHT, &U1_value) == 0){// Lettura corretta dei dati

                                if(get_adc_value(N710_TEMP, &U2_value) == 0){

                                        buf->buf[buf->buf_end++] = RISPONSE;
                                        buf->buf[buf->buf_end++] = (U1_value >> 8);
                                        buf->buf[buf->buf_end++] = (uint8_t) U1_value;

                                        buf->buf[buf->buf_end++] = (U2_value >> 8);
                                        buf->buf[buf->buf_end++] = (uint8_t) U2_value;

                                        invio_risponse = 0;

                                }else{
                                        debug("\r\nError: Failed Read Sensor Measure\r\n");
                                }


                        }else{
```

```
				debug("\r\nError: Failed Read Sensor Measure\r\n");
			}



			if (socket_sendto(broadcast_socket, &dirsens, buf) == pdTRUE) {
				debug("\r\nSend OK ");






			}else{
				debug("\r\nSEND FAILED\r\n");
			}


		}else{
			debug("\r\nError: socket_buffer_get(). Any buffer created\r\n");
		}
	}


	/* ping response handling */
	if ((xTaskGetTickCount() - ping_start)*portTICK_RATE_MS > 1000 && ping_active)
	{
		debug("Ping timeout.\r\n");
		stop_ping();
		if(echo_result.count)
		{
			debug("Response: \r\n");
			for(i=0; i<echo_result.count; i++)
			{
				debug_address(&(echo_result.result[i].src));
				debug(" ");
				debug_int(echo_result.result[i].rssi);
				debug(" dbm, ");
				debug_int(echo_result.result[i].time);
				debug(" ms\r\n");
			}
			echo_result.count=0;
			/*¿Como fare per liberare la memoria di una variabile tipo echo_result?*/
		}
		else
		{
			debug("No response.\r\n");
		}
		ping_active = 0;
	}

	/* stack events */
	if(stack_event)
	{
		buffer_t *buffer = waiting_stack_event(10);
		if(buffer)
		{
			switch (parse_event_message(buffer))
			{
```

```
						case BROKEN_LINK:
							debug("Route

 broken to ");

							debug("\r\n");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						case NO_ROUTE_TO_DESTINATION:
							debug("ICMP message back, no route ");
							debug("\r\n");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						case TOO_LONG_PACKET:
							debug("Payload Too Length\r\n");
							break;

						case DATA_BACK_NO_ROUTE:
							debug("DATA back, No route");
							debug("\r\n");
							debug("To ");
							debug_address(&(buffer->dst_sa));
							debug("\r\n");
							break;

						default:

							break;
					}
					if(buffer)
					{
						socket_buffer_free(buffer);
						buffer = 0;
					}
				}
			} /*end stack events*/
	} /*end main loop*/

}


int8_t get_adc_value(adc_input_t channel, uint16_t *value)
{
	int8_t retval;

	if (adc_convert_single(channel, ADCREF_AVDD, ADCRES_14BIT) == 0)
	{
		retval = 0;
		while (retval != 1)
		{
			retval = adc_result_single(value);
		}
		retval = 0;
	}
	else
```

```
	{
		retval = -1;
	}

	return retval;
}
```