



**Università degli Studi dell'Aquila**

**Centro di Eccellenza DEWS**

*Design Methodologies for Embedded controllers, Wireless interconnect and System-on-chip*

# Wireless Sensor Networks

Ing. Luigi Pomante (PhD)

*luigi.pomante@univaq.it*

## Sommario

- **Parte 1**
  - **Introduzione**
    - Sistemi embedded
    - Sistemi embedded distribuiti
  - **Wireless Sensor Networks**
    - Architettura di sistema
    - Problematiche di progetto
  - **Tecnologie**
    - Hardware
    - Software
  - **Applicazioni**
  - **La nostra esperienza**
  - **Demo**
- **Parte 2**
  - **Esempi nesC/TinyOS**
    - Blink
    - BlinkTask
    - RadioCountToLeds
    - RadioSenseToLeds
    - Oscilloscope
  - **Conclusioni**



*Università degli Studi dell'Aquila*

*Centro di Eccellenza DEWS*

*Design Methodologies for Embedded controllers, Wireless interconnect and System-on-chip*

# Wireless Sensor Networks

*Parte 1*

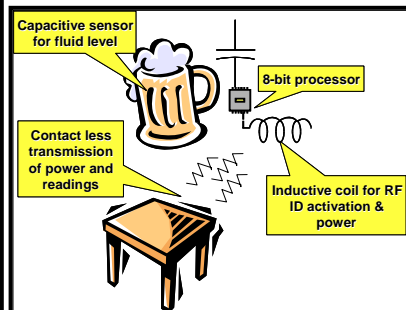
## Introduzione

## Introduzione

- Sistemi embedded
  - Sistemi digitali sviluppati appositamente per una specifica applicazione (*sistemi dedicati*)
    - Spesso “incastonati” in un sistema più ampio
  - Caratteristiche principali
    - Vincoli funzionali e soprattutto non funzionali ne influenzano fortemente l'architettura hw/sw
      - *Time-to-market*, costi, volumi di vendita, prestazioni, consumi energetici, fattore di forma, tempo reale
    - Le risorse hw/sw di sistema sono generalmente limitate, eterogenee e a volte sviluppate *ad-hoc*

## Introduzione

- Sistemi embedded
  - Prodotti tradizionali e servizi innovativi...



## Introduzione



- Sistemi embedded distribuiti
  - Sistemi in cui l'elaborazione e la memorizzazione dei dati sono suddivise equamente tra un numero non esiguo di sottosistemi opportunamente interconnessi
    - Quando la rete d'interconnessione e comunicazione ha una struttura non banale, l'approccio al progetto hw/sw tradizionale deve essere integrato con tecniche specifiche
      - I protocolli di comunicazione diventano un elemento particolarmente rilevante e critico

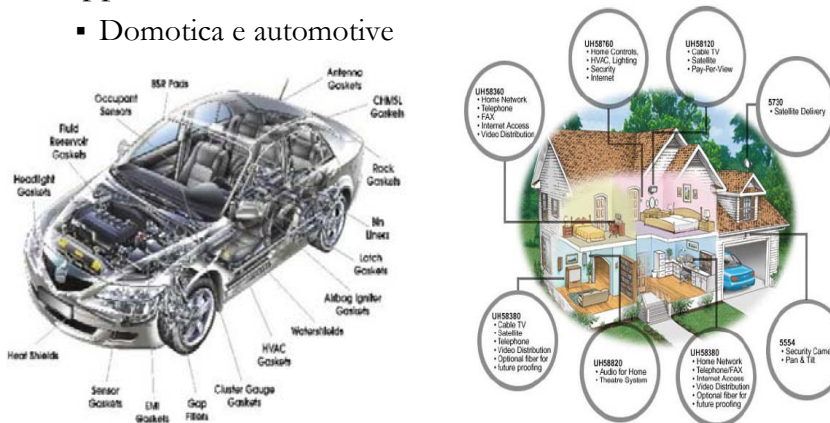
## Introduzione



- Sistemi embedded distribuiti
  - Esistono due classi principali di sistemi che si differenziano in base alla tipologia di interconnessione e di comunicazione
    - Sistemi wired
      - Infrastruttura di comunicazione fissa e nota
    - Sistemi wireless
      - Infrastruttura di comunicazione più flessibile e più complessa

## Introduzione

- Sistemi embedded distribuiti
  - Applicazioni distribuite cablate
    - Domotica e automotive



DEWS

9

## Introduzione

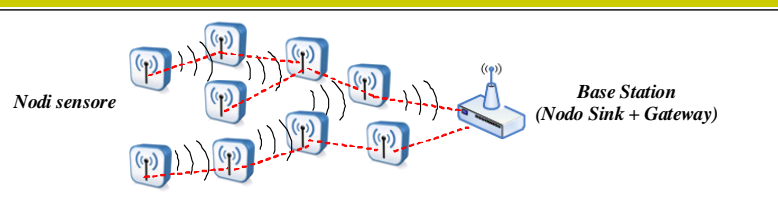
- Sistemi embedded distribuiti
  - Grande interesse stanno riscuotendo sistemi noti come *Sensor Networks*, basati su comunicazione cablata o wireless
    - Questi ultimi, detti *Wireless Sensors Networks*, trovano applicazioni sempre nuove in settori che vanno dalla telemedicina, alla protezione ambientale, al monitoraggio dei beni culturali, ecc...
      - *Ubiquitous/Pervasive Computing*

DEWS

10

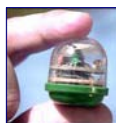
## Wireless Sensor Networks

## Wireless Sensor Networks



- Reti *wireless* formate da dispositivi autonomi, i *nodi sensore*, distribuiti nello spazio e dotati di sensori, che cooperano per fornire una funzione di monitoraggio dell'ambiente circostante

- Elementi chiave



- Nodo sensore (*nodo, mote*) e Base Station
- Comunicazioni *wireless* a corto raggio (*multi-hop*)

## Wireless Sensor Networks

- Ogni nodo integra, oltre ai sensori veri e propri, diversi componenti tra cui un microprocessore (o un microcontrollore), una piccola quantità di memoria, un *transceiver*, un'antenna e una sorgente di energia e, a volte, anche attuatori
  - La dimensione e la potenza di calcolo di un nodo variano significativamente da nodi complessi e potenti, delle dimensioni di un portatile, fino a nodi estremamente semplici e piccoli



WeC (1999)



Rene (2000)



Dot (2001)



MICA (2002)



Speck (2003)



Telos (2004)

DEWS

13

## Wireless Sensor Networks

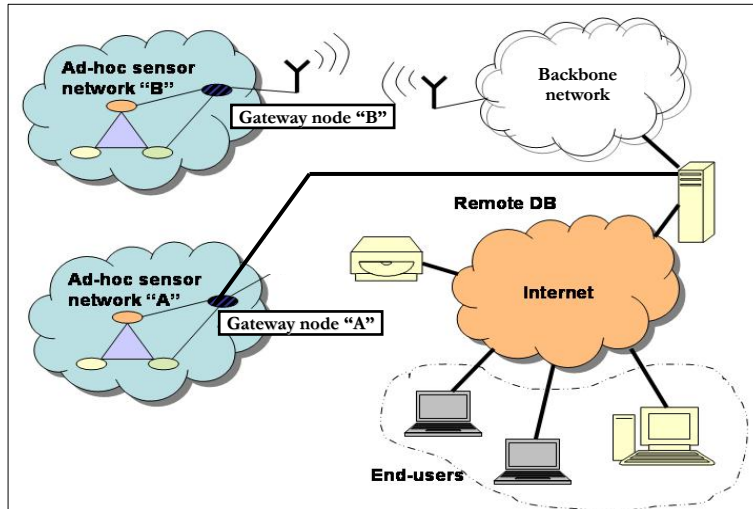
- A prescindere dalla specifica realizzazione e dai singoli componenti utilizzati per i nodi, le WSN presentano caratteristiche peculiari e distintive
  - Autoconfigurazione
  - Scalabilità
  - Robustezza
  - Flessibilità
  - Basso costo
  - Efficienza energetica

DEWS

14

## Wireless Sensor Networks

- Architettura di sistema



DEWS

15

## Wireless Sensor Networks

- Problematiche di progetto

- Modello del sistema
  - Nodi fisici vs. Componenti funzionali
  - Computazione locale vs. Comunicazione
- Architettura hardware dei nodi
  - Microprocessore/Microcontrollore
    - IBM 8051, Atmel ATmega128L, XScale PXA271, TI MSP430,...
  - Chipset per la comunicazione e la corrispondente antenna
    - ChipCon CC1100 e CC2420
  - Bus di comunicazione locale
    - SPI, I2C o proprietari

DEWS

16

## Wireless Sensor Networks



- Problematiche di progetto
  - Architettura software dei nodi
    - Data la necessità di stack protocollari, timer, allarmi, input/output, interruzioni, è molto comune dover utilizzare un sistema operativo
      - TinyOS, MANTIS, FreeRTOS, SOS, Contiki
  - Modello di programmazione
    - Il modello di programmazione di una rete di sensori fornisce un supporto all'accesso ai dati e alle risorse di calcolo della rete
      - A oggi, il modello più comune è quello delle basi di dati distribuite

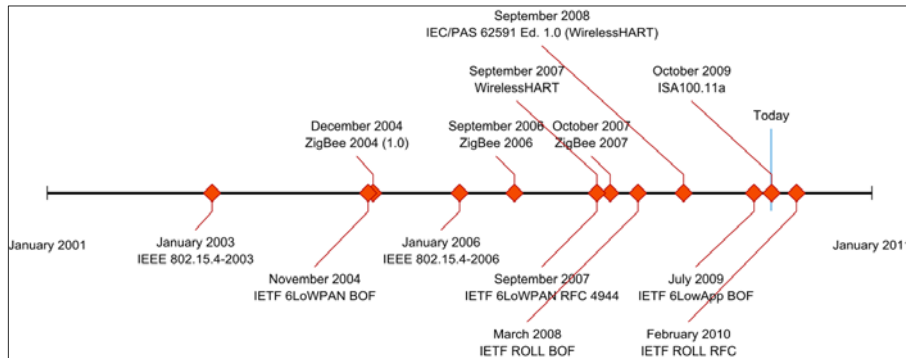
## Wireless Sensor Networks



- Problematiche di progetto
  - Protocolli di comunicazione
    - I protocolli di comunicazione impiegati nelle WSN sono diversi ed eterogenei
      - Ai livelli più bassi, ci si appoggia a protocolli standard
        - IEEE 802.15.4, SP-100.11a, 6lowpan, Bluetooth, ecc...
    - Un aspetto essenziale è la riconfigurabilità dinamica della topologia della rete che, in molti casi reali, è soggetta a cambiamenti
      - Specifici algoritmi di routing per WSN
        - SPIN (Sensor Protocols for Information via Negotiation), Directed Diffusion, Rumor Routing, Q-RC (Q-learning Routing and Compression), ecc...

# Wireless Sensor Networks

- Problematiche di progetto: protocolli

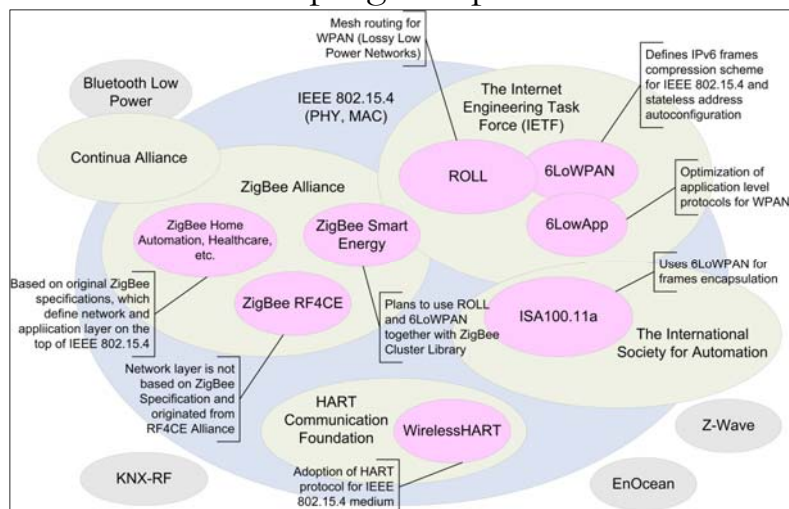


DEWS

19

# Wireless Sensor Networks

- Problematiche di progetto: protocolli



DEWS

20

## Wireless Sensor Networks



- Problematiche di progetto
  - Risparmio energetico
    - Nodo
      - Progettazione componenti e architettura hw/sw
      - Meccanismi per la (auto)gestione del consumo energetico
    - Rete
      - Protocolli *energy-aware*
    - Sistema
      - Applicazioni *energy-aware*

## Wireless Sensor Networks



- Problematiche di progetto
  - Affidabilità
    - Nodo
      - Ridondanza strutturale, temporale e di informazione
      - Gestione delle anomalie
    - Rete
      - Ridondanza spaziale, temporale e di informazione
      - Gestione delle anomalie
    - Sistema
      - Gestione delle anomalie

## Wireless Sensor Networks



- Problematiche di progetto
  - Middleware
    - Nodo
      - Interfacce & Macchine virtuali
  - Rete
    - Gestione eterogeneità piattaforme (radio, hw e sistemi operativi)
    - Gestione dei servizi
    - Macchine virtuali
  - Sistema
    - Metodologie e linguaggi di programmazione

## Wireless Sensor Networks



- Problematiche di progetto
  - Servizi: sincronizzazione
    - Nodo
      - Gestione riferimenti interni
  - Rete
    - Algoritmi distribuiti per la gestione dei riferimenti esterni
  - Sistema
    - Analisi ed elaborazione dei dati

## Wireless Sensor Networks



- Problematiche di progetto
  - Servizi: localizzazione
    - Nodo (*Ranging*)
      - Tecnologie radio (interferometria, RSSI), ultrasuoni, laser
    - Rete (*Positioning*)
      - Algoritmi distribuiti ad approssimazioni successive
      - Integrazione con riferimenti esterni
    - Sistema
      - Integrazione con GIS

## Wireless Sensor Networks



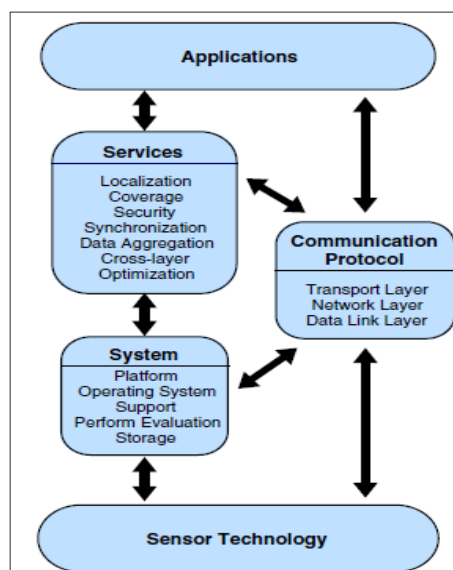
- Problematiche di progetto
  - Servizi: codifica distribuita
    - Nodo
      - Analisi ed elaborazione correlazioni
    - Rete
      - Algoritmi distribuiti
        - Elezione, clustering
      - Analisi ed elaborazione dei dati
    - Sistema
      - Analisi ed elaborazione dei dati

## Wireless Sensor Networks

- Problematiche di progetto
  - Servizi: sicurezza
    - Nodo
      - Algoritmi di crittografia
  - Rete
    - Sistemi crittografici
    - *Intrusion Detection & Monitoring Systems*
  - Sistema
    - Sistemi crittografici

## Wireless Sensor Networks

- Problematiche di progetto
  - Visione d'insieme



# Wireless Sensor Networks



- Problematiche di progetto
  - Metodologie e strumenti per WSN-CAD
    - Analisi & Specifica Requisiti
      - Notazioni, linguaggi e formalismi
    - Progetto: “Configuratore”
    - Verifica & Validazione
      - Simulazione
        - Modelli (nodo, rete): energetici, prestazionali, etc.
      - *In-Network Debug*
    - Implementazione
      - Supporto alla generazione di codice

## Tecnologie

Hardware

## Tecnologie HW



- Nodo sensore
  - Elaborazione, Comunicazione
- Sensor board
  - Acquisizione
- Programming Board
  - Programmazione nodi
- Base Station (Nodo sink + Gateway)
  - Interfacciamento e gestione WSN
- Panoramica

## Tecnologie HW

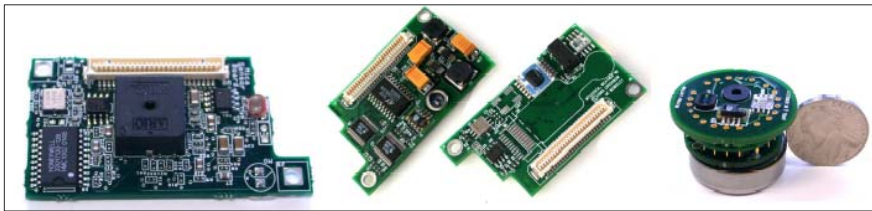


- Nodo sensore: XBOW *MicaZ*
  - Elaborazione: microcontrollore Atmel ATmega128L
    - MPU: 8-bit RISC (0-8 MHz)
    - Memoria
      - Programma: 128K Bytes Flash
      - Lavoro: 4K Bytes SRAM
    - ADC, UART, GPIO, I2C, SPI, Timer
  - Comunicazione: *Transceiver* Chipcon CC2420
    - IEEE 802.15.4 (2,4 GHz, 250 kbps, range 20-100 m)
  - Memorizzazione locale: Flash 512 KB



## Tecnologie HW

- Sensor board
  - Differenti tipologie di “sensori”
    - Luce, temperatura, pressione, umidità
    - Accelerometri, magnetometri, misuratori di distanza
    - Microfono, foto/videocamera, ricevitori GPS



## Tecnologie HW

- Base Station + Programming Board
  - Collegamento *wired* PC-nodo (*wireless* con altri nodi)
    - Parallela, seriale (MIB 510/520), ethernet



## Tecnologie HW



- Panoramica

NODE TYPE	SAMPLE AND SIZE	APPLICATION	MEMORY	OTHER
SPECIALIZED SENSING PLATFORM	SPEC (mm <sup>3</sup> )	RF tag or specialized sensor	3K RAM	
GENERIC SENSING PLATFORM	MOTE (1-10 c m <sup>3</sup> )	General purpose sensor and commun.	4K RAM 128K FLASH	TinyOS
HIGH-BANDWIDTH SENSING	IMOTE (1-10 c m <sup>3</sup> )	High bandwidth sensor (video, acoustic, etc.)	64KB RAM 512KB FLASH	TinyOS, BLUETOOTH, CONNECTIVITY WITH CELL PHONES
GATEWAY	STARGATE (> 10 c m <sup>3</sup> )	High bandwidth sensor plus gateway	<512KB RAM <32MB FLASH	LINUX OR WINDOWS, SERIAL CONNECTION TO SENSOR NETWORK

DEWS

35

## Tecnologie

Software

## Tecnologie SW



- Requisiti generali
- TinyOS World
  - nesC (*Networked Embedded Systems C*)
    - TinyOS (2.x)
  - XNP/Deluge
    - *In-Network Reprogramming*
  - TinyDB
    - *Networked Distributed Query Processing*
- Panoramica

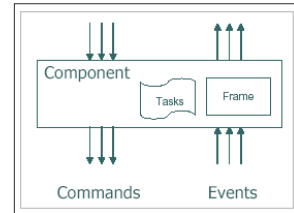
## Tecnologie SW



- Requisiti generali
  - Dimensioni (KB) contenute
    - Sia per il codice (*footprint*) sia per i dati
  - Utilizzo efficiente delle risorse HW
    - In particolare in relazione ai consumi energetici
  - Parallelismo a grana fine
    - Almeno a livello di *thread*
  - Alta modularità
    - “Portarsi dietro” solo il necessario
  - Ambienti di sviluppo flessibili e *open-source*

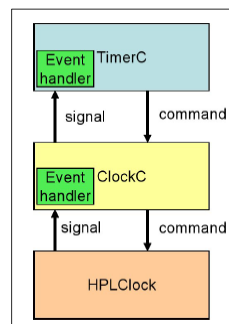
## Tecnologie SW

- nesC
  - Estensione linguaggio C per *networked embedded systems*
    - Programmazione per componenti
      - Moduli e Interfacce
      - Configurazioni
      - Collegamenti
    - Programmazione orientata agli eventi
      - Comandi utilizzati/offerti
      - Eventi segnalati/gestiti
        - Corrispondono ad interruzioni hw o sw



## Tecnologie SW

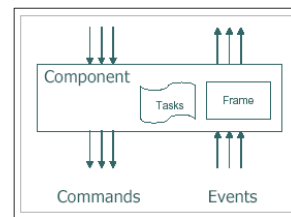
- nesC
  - I componenti utilizzatori sono collegati (*wired*) ai fornitori
    - Si crea una gerarchia di componenti
  - I comandi
    - Vanno verso il basso
    - Ritornano al chiamante
  - Gli eventi
    - Vanno verso l'alto
    - Ritornano al segnalatore



## Tecnologie SW



- nesC
  - Oltre a comandi/eventi esiste il concetto di *task* che introduce due livelli di priorità nell'esecuzione del codice
    - Eventi
      - Codice ad alta priorità
    - Task
      - Codice a bassa priorità



## Tecnologie SW



- nesC
  - Task
    - Computazione medio-lunga in background
    - Atomici rispetto ad altri task
    - Interrotti (*preemption*) dagli eventi
  - Eventi
    - Alta reattività
    - Breve durata (demandano lavoro ai task)
  - Task ed eventi possono richiamare comandi che tipicamente danno luogo ad una esecuzione di tipo *split-phase*
    - Questo permette un parallelismo a grana fine

## Tecnologie SW



- nesC
  - Es. Componente di tipo Modulo

```
module XYZ1
{
  provides interface Interface1 as I1;
  provides interface Interface2;
  ...
  uses interface Interface3 as I3;
  uses interface Interface2;
  ...
}
implementation
{
  command void I1.cmd1() {
    ...
  }

  event void Interface2.ev1() {
    ...
  }
}
```

## Tecnologie SW



- nesC
  - Es. Componente di tipo Configurazione

```
configuration XYZ
{
  ...
}
implementation
{
  components XYZ1, XYZ2;

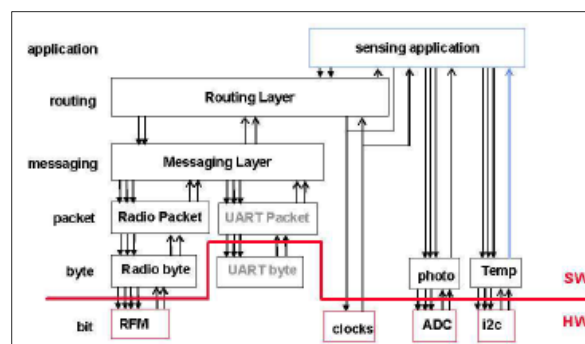
  ...
  XYZ1.Interface1 -> XYZ2.Interface1;
  XYZ1.Interface2 <- XYZ2;
  ...
}
```

## Tecnologie SW

- TinyOS ([www.tinyos.net](http://www.tinyos.net))
  - Si tratta di una libreria di componenti nesC che offre alcune funzionalità tipiche di un sistema operativo
    - Scheduler
    - Driver
      - Componenti per leggere dati da sensori
      - Componenti per inviare comandi ad attuatori
      - Componenti per gestire le comunicazioni radio
    - Power Management
      - Mantenere i dispositivi HW nel più basso stato di potenza dissipata possibile
    - No concetti di *kernel*, processi, gestione della memoria

## Tecnologie SW

- TinyOS
  - Un applicazione è quindi un grafo di componenti
    - Componenti di S.O. + Componenti Applicativi
      - Il tutto è compilato e usato per programmare il singolo nodo



## Tecnologie SW



- TinyOS
  - Per ogni applicazione esiste inoltre la configurazione *top-level* che contiene il componente *MainC*
    - Fornisce i servizi base di TinyOS ( $\approx 200$  Bytes)

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  ...

  BlinkC -> MainC.Boot;
  ...
  BlinkC.Timer0 -> Timer0;
  ...
  BlinkC.Leds -> LedsC;
}
```

## Tecnologie SW



- TinyOS
  - La filosofia di sviluppo di una applicazione è del tipo
    - ***Hurry Up and Sleep!!!***
      - Ovvero cercare di rimanere nello stato di *sleep* il più possibile per risparmiare energia
      - Quando il nodo è risvegliato da un evento, cercare di eseguire il lavoro associato il più in fretta possibile e poi tornare a dormire
        - *Interrupt-driven & Split-phase*

## Tecnologie SW



- XNP/Deluge
  - La riprogrammazione diretta dei nodi può essere molto onerosa
    - Una volta che una WSN è stata posizionata questa potrebbe non essere più facilmente raggiungibile
    - I nodi possono essere così tanti da richiedere un tempo eccessivamente lungo
  - La soluzione al problema è la
    - ***In-Network Reprogramming***

## Tecnologie SW



- XNP/Deluge
  - Consiste nel riprogrammare un nodo via radio sovrascrivendo (*re-flashing*) le istruzioni in memoria
    - La XNP è implementata come un servizio separato
    - Modificare le applicazioni (una volta) per abilitarle alla XNP non è particolarmente oneroso
      - Aggiunta di particolari componenti
  - La XNP prevede tre fasi:
    - *Download*
    - *Query*
    - *Reprogram*

## Tecnologie SW



- TinyDB
  - La gestione dei dati provenienti da una (o più) WSN può essere di due tipologie principali
    - Tradizionale (procedurale)
      - Esplicita richiesta dei dati dai singoli sensori
      - Dati elaborati in modo centralizzato
    - *DB-style* (dichiarativo)
      - Richiesta di informazioni tramite *query*
      - Elaborazione implicita e distribuita dei dati all'interno della rete
      - All'utente arrivano solo le informazioni di interesse

## Tecnologie SW



- TinyDB
  - Sistema di elaborazione distribuita di query per estrarre informazioni da una WSN basata su TinyOS
    - Query espresse tramite una variante di SQL (TinySQL)
      - Le query sono inoltrate da PC ai nodi e possono essere memorizzate localmente ed elaborate periodicamente
    - I dati raccolti dai nodi, sono opportunamente elaborati (filtraggio, aggregazione, ecc.) nel percorso verso il PC
      - Gli algoritmi *low-power* per l'elaborazione e il routing sono completamente mascherati all'utente finale

## Tecnologie SW



- TinyDB
  - Si basa su una tabella che contiene informazioni relative ai singoli nodi e ai sensori disponibili su ciascuno di essi
    - Tabella *Sensors*

Node_ID	Light	Temp	...
1	28	12	...
2	55	16	...
3	48	11	...

## Tecnologie SW



- TinyDB
  - Esempi di query

```
SELECT NodeID, Light
FROM Sensors
EPOCH DURATION 5min
```

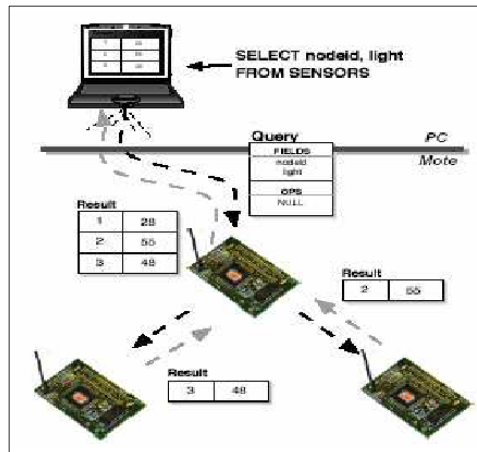
```
SELECT AVG(Light), AVG(Temp)
FROM Sensors
WHERE NodeID>1
EPOCH DURATION 5min
```

```
ON event:
SELECT NodeID, Light, Temp
FROM Sensors
WHERE Light>Threshold
TRIGGER ACTION SetSnd(512)
```

```
SELECT AVG(Temp)
FROM Sensors
GROUP BY Light
HAVING AVG(Temp)>10
EPOCH DURATION 5min
```

## Tecnologie SW

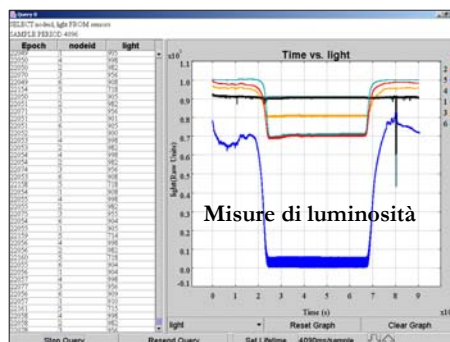
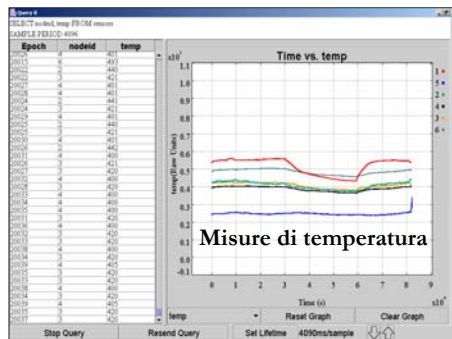
- TinyDB
  - Un sguardo dietro la tabella...
    - Logicamente centralizzata nel PC
    - Fisicamente partizionata e distribuita sui nodi



DEWS

## Tecnologie SW

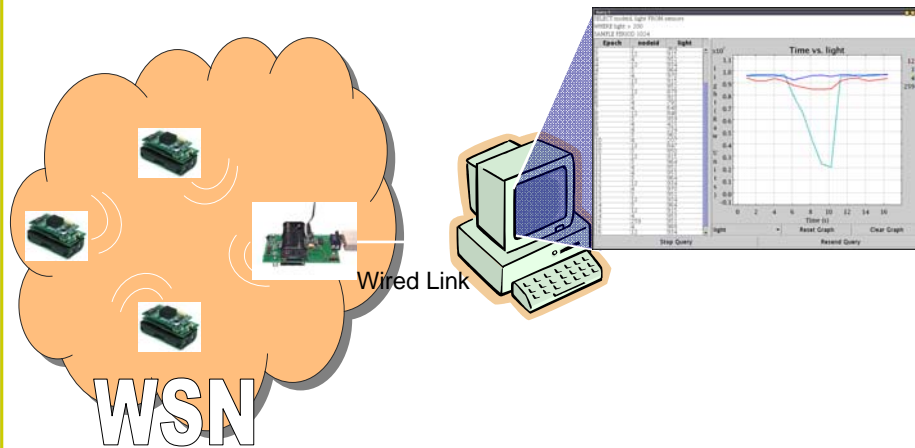
- TinyDB al lavoro...



DEWS

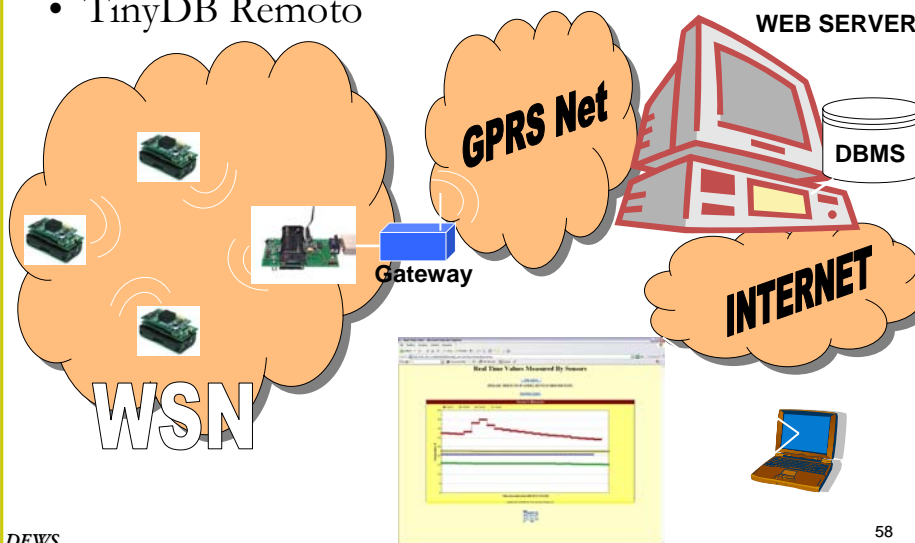
## Tecnologie SW

- TinyDB Locale



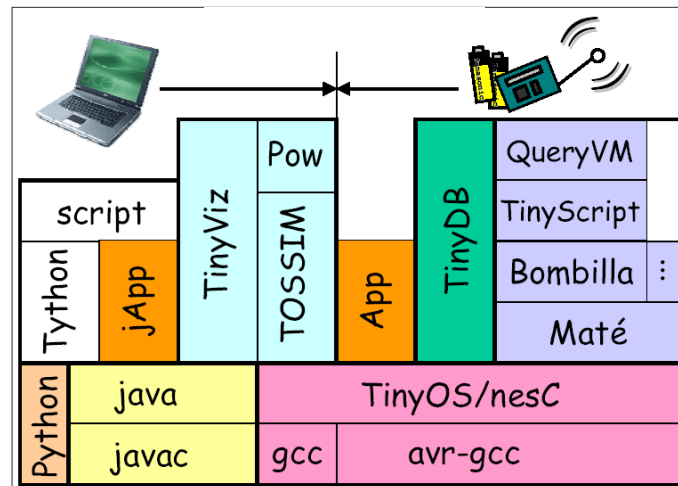
## Tecnologie SW

- TinyDB Remoto



# Tecnologie SW

- Panoramica



DEWS

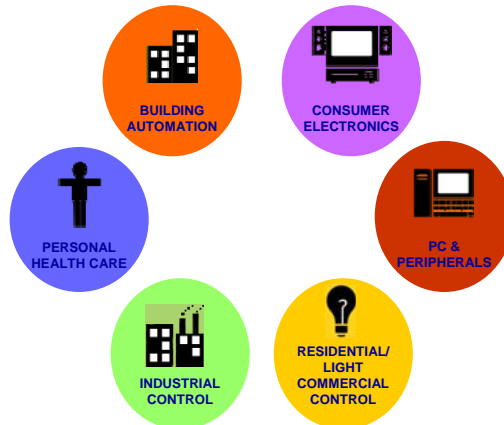
59

## Applicazioni

## Applicazioni

- Svatiati campi applicativi

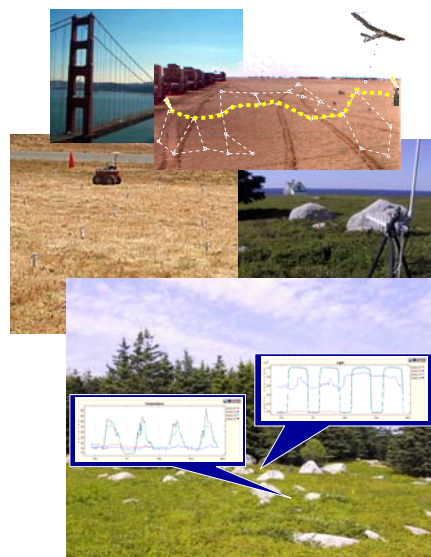
- Militari
- Ambientali
- Mediche
- Domotiche
- Commerciali
- Industriali
- Civili
- ...



## Applicazioni

- Applicazioni Ambientali

- Due categorie principali
  - Monitoraggio dell'habitat
  - Monitoraggio di strutture
- Esempi
  - Rilevamento incendi/alluvioni
  - Monitoraggio frane
  - Agricoltura di precisione
  - Monitoraggio qualità acqua/aria
  - Monitoraggio animali protetti
  - Monitoraggio ponti/edifici



## Applicazioni

- Applicazioni Ambientali

- Esempio famoso

- Great Duck Island

- Piccola isola degli USA, Maine

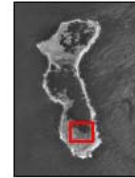
- Studio sui Petrel, una particolare specie di uccello marino

- 2002

- Disposizione di 32 mote con vari sensori

- Monitoraggio del microclima attorno ai nidi

- Informazioni veicolate verso la rete Internet in tempo reale



## Applicazioni

- Applicazioni Mediche

- Esempi

- Monitoraggio continuo di pazienti

- In ambulatorio o in ospedale

- In casa per pazienti cronici o anziani

- Adattamento dell'ambiente al paziente

- Raccolta di dati clinici



## Applicazioni

- Applicazioni Domotiche
  - Esempi
    - Gestione locale e remota della casa
      - Automazione degli elettrodomestici
      - Luminosità
        - Interruttori wireless
      - Temperatura
        - Termostati wireless
    - Controllo della casa



## Applicazioni

- Applicazioni Commerciali
  - Tracking
    - Rilevamento della posizione e del movimento
    - Controllo del traffico
  - Controllo dell'ambiente in ufficio
    - Temperature
    - Disponibilità stanze (Intel)
  - Museo interattivo
    - Interazione con oggetti esposti
    - Localizzazione nel museo



## Applicazioni

- Applicazioni Militari
  - Monitoraggio di attrezzature e munizioni
  - Sorveglianza del campo di battaglia
    - Tipico ambiente ostile
  - Riconoscimento del tipo di attacco
    - Nucleare, biologico, chimico

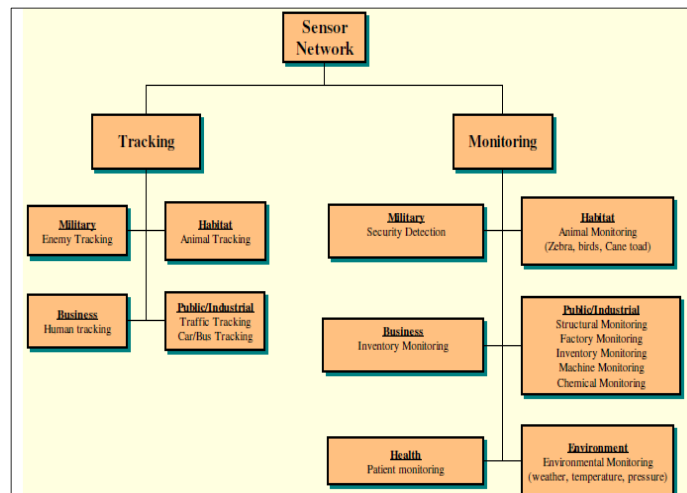


DEWS

67

## Applicazioni

- Classificazione



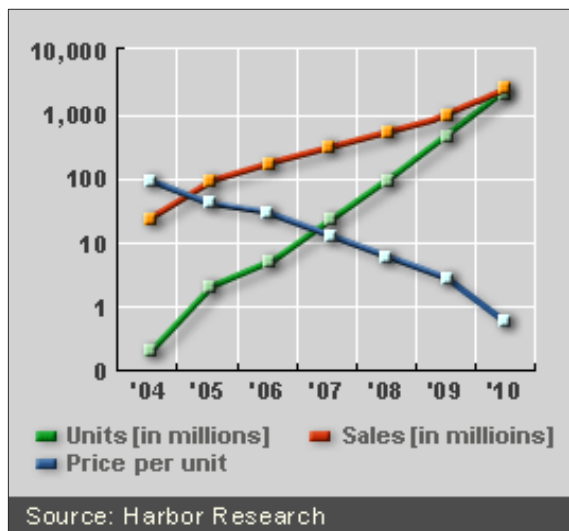
DEWS

68

## Applicazioni



- Costi
  - Previsioni per i singoli nodi



La nostra esperienza

## La nostra esperienza

- **Ricerca**

- Centro di Eccellenza per la Ricerca DEWS

- **DEWS Lab**

- **EECI Lab**

- European Embedded Control Institute

- **Innovazione**

- WEST Aquila Srl

- Spin-off del Dipartimento di Elettrica e Informazione dell'Università degli Studi di L'Aquila

## La nostra esperienza

### ***XBOW/Memsic Family***

**HW:** Mica2/MicaZ/Iris, Stargate, eKo, sensor boards

**SW:** nesC, Java, TinyOS, Linux

**Radio:** IEEE802.15.4/ZigBee, Wi-Fi, GPRS/UMTS



## La nostra esperienza

### **Texas Instruments Family**

**HW:** CC2430/CC2431

**SW:** C, C++, Java, Linux/Windows

**Radio:** IEEE802.15.4/ZigBee



DEWS

73

## La nostra esperienza

### **Miscellanea**

**HW:** Siemens XT75, Radiocrfts RC2300AT, Shimmer Research – Shimmer Node, WEST Node & Sensor board

**SW:** nesC, C/C++, java, TinyOS, Linux/Windows, WEB

**Radio:** IEEE802.15.4/ZigBee, BlueTooth, GPRS/UMTS

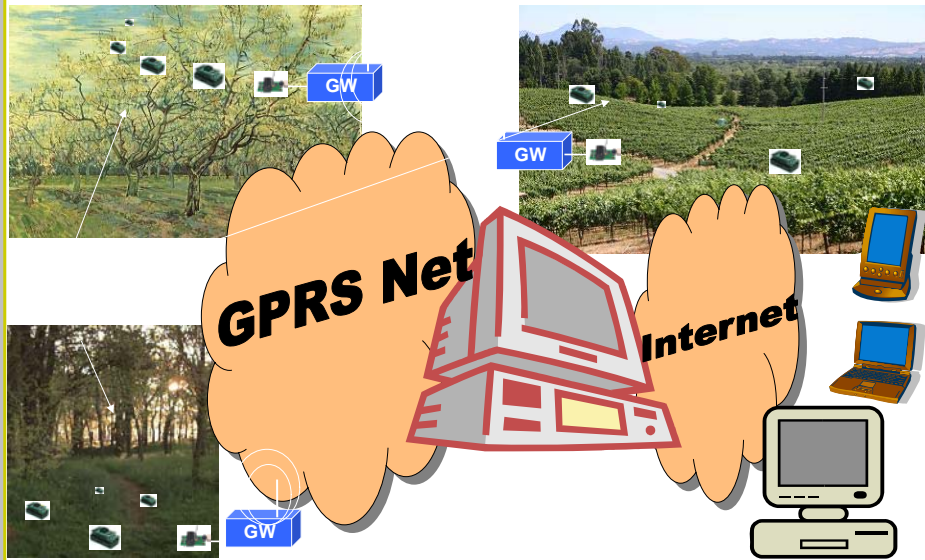


DEWS

74

# La nostra esperienza

Web Sensors Network



DEWS

75

# La nostra esperienza

WERISE&TRACK

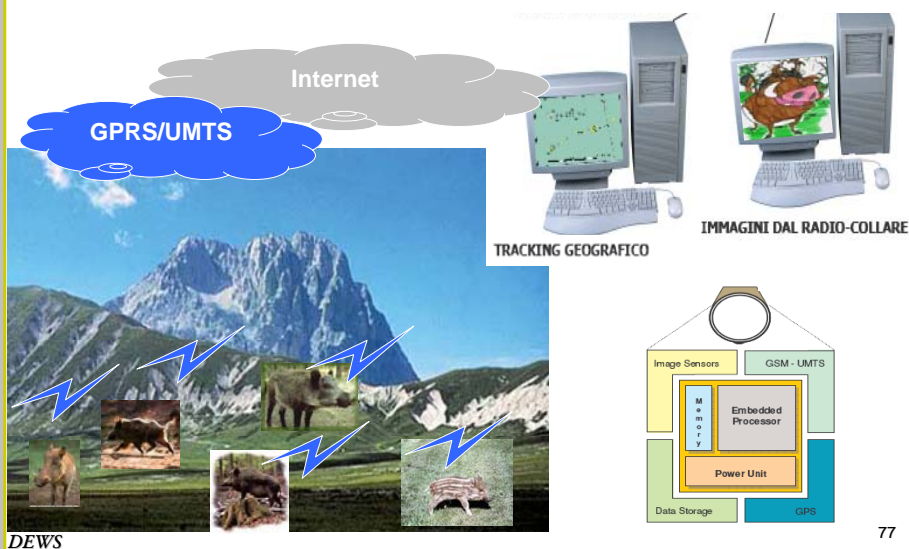


DEWS

76

# La nostra esperienza

Wild Animals Tracking And Monitoring



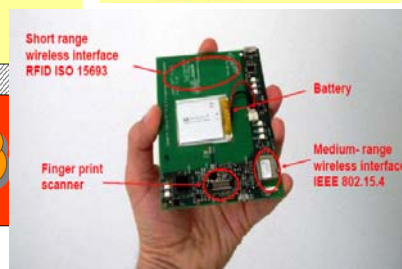
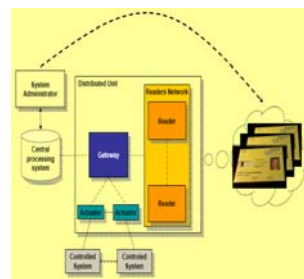
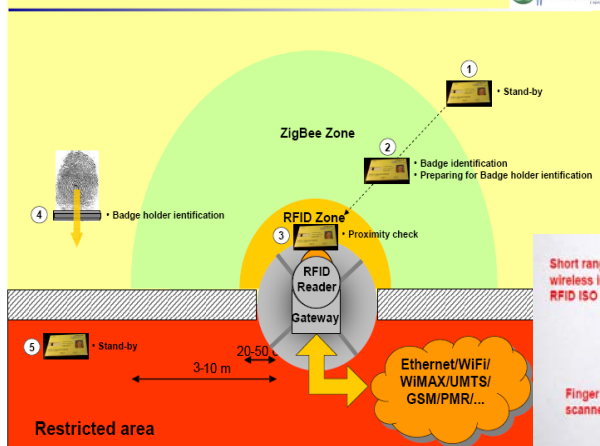
77

# La nostra esperienza

Piattaforma di autenticazione biometrica: controllo accessi aree riservate



XGW – Access phases

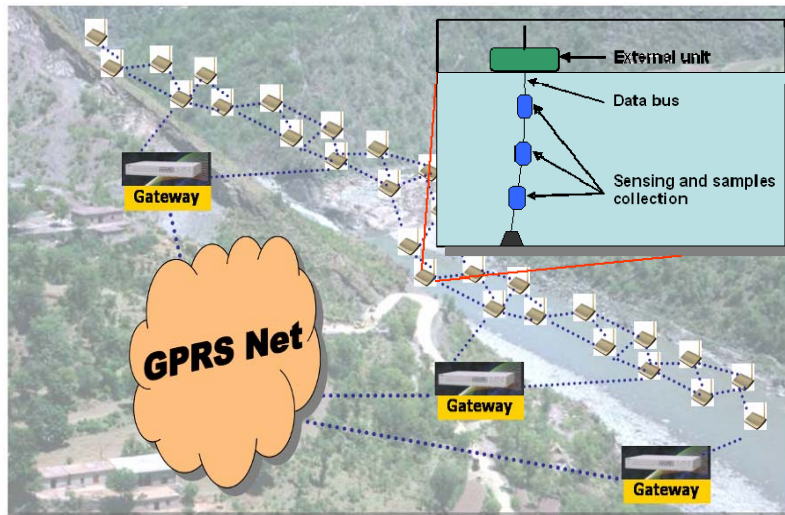


DEWS

78

## La nostra esperienza

Monitoraggio qualità acque

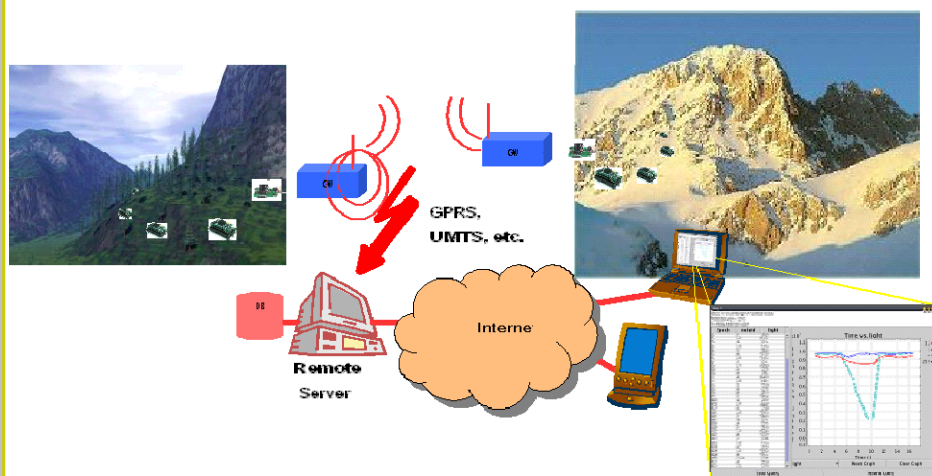


DEWS

79

## La nostra esperienza

Monitoraggio frane

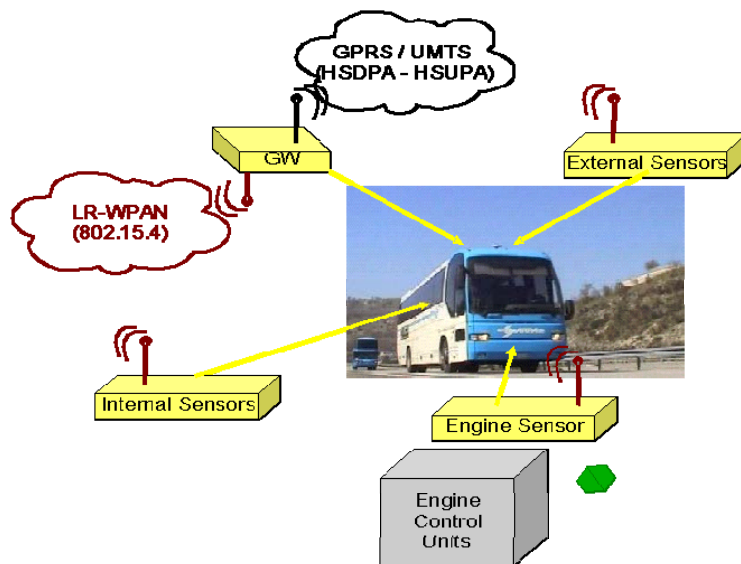


DEWS

80

# La nostra esperienza

Monitoraggio flotte



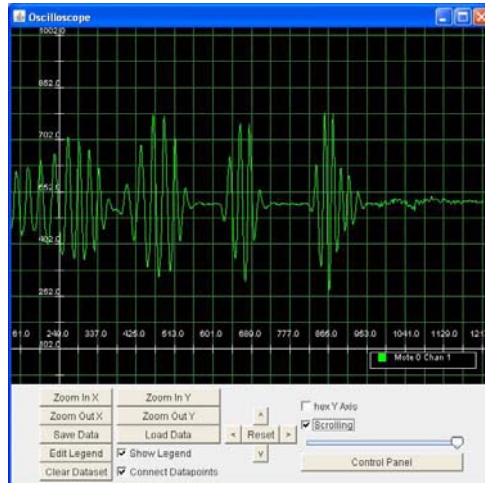
DEWS

81

## Demo

## Demo

- TinyOS 2.x
  - *Oscilloscope*
    - *Nesc/TinyOS*
    - *Java/Linux*



DEWS

83



**Università degli Studi dell'Aquila**

**Centro di Eccellenza DEWS**

*Design Methodologies for Embedded controllers, Wireless interconnect and System-on-chip*

## Wireless Sensor Networks

*Parte 2*

## Esempi nesC/TinyOS

Blink (TinyOS 2.x)

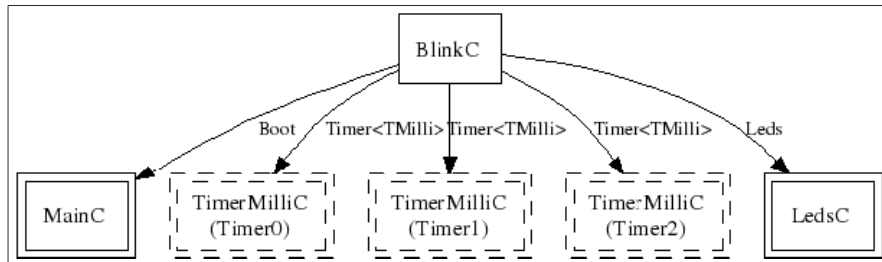
### Blink



- Fa lampeggiare i 3 led di un nodo sensore
  - I led lampeggiano a frequenze di 1Hz, 2Hz, e 4Hz
- Componenti applicativi
  - *BlinkAppC (Configuration)*
  - *BlinkC (Module)*
- Componenti di sistema
  - *MainC, LedsC, TimerMilliC*

## Blink

- BlinkAppC: grafo dei componenti



	Singleton	Generic
Module		
Configuration		

DEWS

87

## Blink

- BlinkAppC.nc

```
configuration BlinkAppC
{
}
implementation
{
    components MainC, BlinkC, LedsC;
    components new TimerMilliC() as Timer0;
    components new TimerMilliC() as Timer1;
    components new TimerMilliC() as Timer2;

    BlinkC -> MainC.Boot;

    BlinkC.Timer0 -> Timer0;
    BlinkC.Timer1 -> Timer1;
    BlinkC.Timer2 -> Timer2;
    BlinkC.Leds -> LedsC;
}
```

DEWS

88

## Blink



- BlinkC.nc

```
#include "Timer.h"

module BlinkC
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
```

## Blink



- BlinkC.nc

```
event void Boot.booted()
{
  call Timer0.startPeriodic( 250 );
  call Timer1.startPeriodic( 500 );
  call Timer2.startPeriodic( 1000 );
}
```

## Blink

- BlinkC.nc

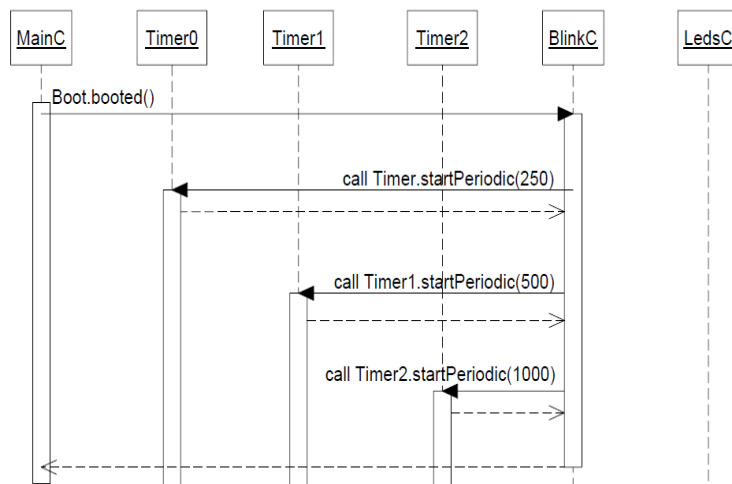
```
event void Timer0.fired()
{
    dbg("BlinkC", "Timer 0 fired @ %s.\n", sim_time_string());
    call Leds.led0Toggle();
}

event void Timer1.fired()
{
    dbg("BlinkC", "Timer 1 fired @ %s \n", sim_time_string());
    call Leds.led1Toggle();
}

event void Timer2.fired()
{
    dbg("BlinkC", "Timer 2 fired @ %s.\n", sim_time_string());
    call Leds.led2Toggle();
}
```

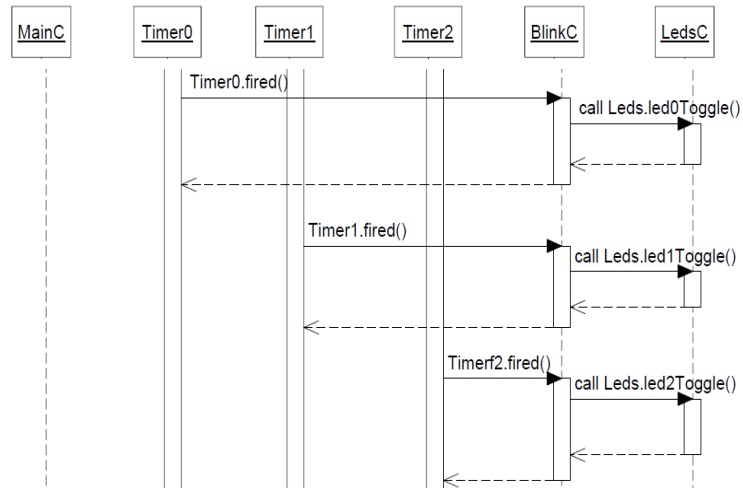
## Blink

- Sequence diagram: INIT



## Blink

- Sequence diagram: attività periodica



DEWS

93

## Esempi nesC/TinyOS

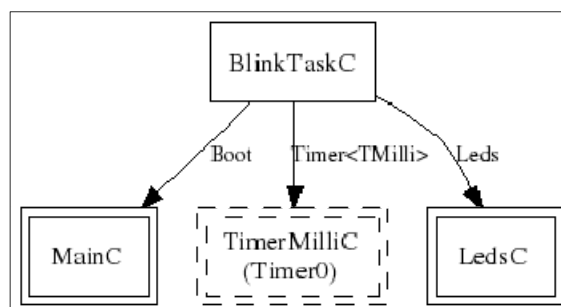
BlinkTask (TinyOS 2.x)

## BlinkTask

- Fa lampeggiare un led del nodo per mezzo di un task TinyOS
- Componenti applicativi
  - *BlinkTaskAppC* (Configuration)
  - *BlinkTaskC* (Module)
- Componenti di sistema
  - *MainC*, *LedsC*, *TimerMilliC*

## BlinkTask

- BlinkTaskAppC: grafo dei componenti



## BlinkTask



- BlinkTaskAppC.nc

```
configuration BlinkAppC
{
}
implementation
{
    components MainC, BlinkTaskC, LedsC;
    components new TimerMilliC() as Timer0;

    BlinkTaskC -> MainC.Boot;
    BlinkTaskC.Timer0 -> Timer0;
    BlinkTaskC.Leds -> LedsC;
}
```

## BlinkTask



- BlinkTaskC.nc

```
#include "Timer.h"

module BlinkTaskC
{
    uses interface Timer<TMilli> as Timer0;
    uses interface Leds;
    uses interface Boot;
}
implementation
{
    task void toggle()
    {
        call Leds.led0Toggle();
    }
}
```

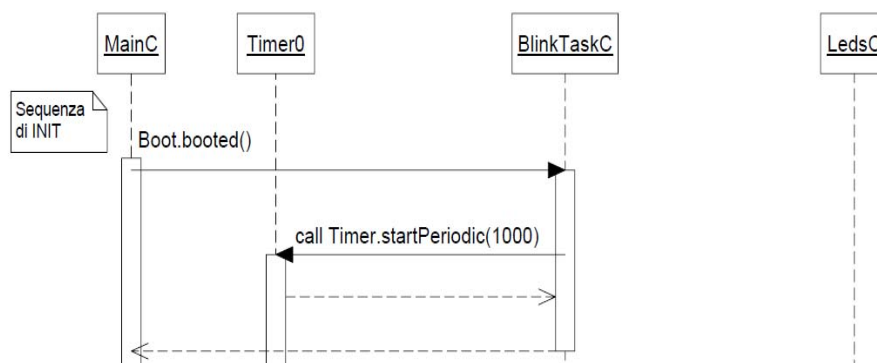
## BlinkTask

- BlinkTaskC.nc

```
event void Boot.booted()  
{  
  call Timer0.startPeriodic( 1000 );  
}  
  
event void Timer0.fired()  
{  
  post toggle();  
}  
}
```

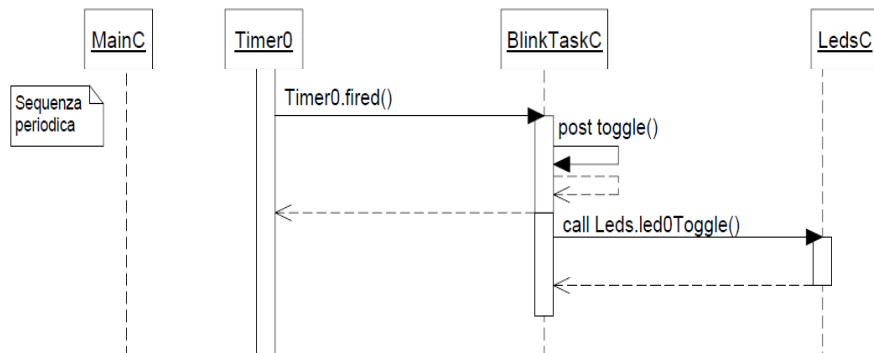
## BlinkTask

- Sequence diagram: INIT



## BlinkTask

- Sequence diagram: attività periodica



## Esempi nesC/TinyOS

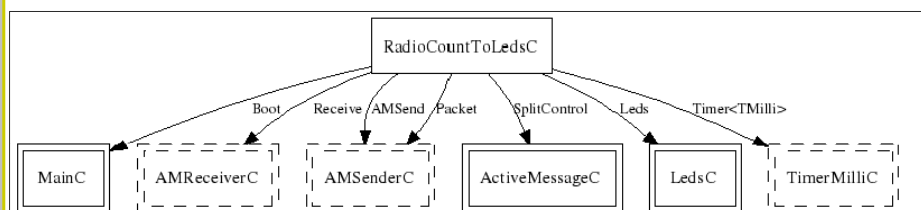
RadioCountToLeds (TinyOS 2.x)

## RadioCountToLeds

- Incrementa periodicamente (4Hz) un contatore e ne spedisce il valore via radio in broadcast
  - I nodi che ricevono visualizzano i tre LSB sui led
- Componenti applicativi
  - *RadioCountToLedsAppC (Configuration)*
  - *RadioCountToLedsC (Module)*
- Componenti di sistema
  - *MainC, LedsC, TimerMilliC*
  - *AMReceiverC, AMSenderC, ActiveMessageC*

## RadioCountToLeds

- RadioCountToLedsAppC: grafo dei componenti



## RadioCountToLeds



- RadioCountToLeds.h

```
typedef nx_struct radio_count_msg {  
    nx_uint16_t counter;  
} radio_count_msg_t;  
  
enum {  
    AM_RADIO_COUNT_MSG = 6,  
};
```

## RadioCountToLeds



- RadioCountToLedsAppC.nc

```
#include "RadioCountToLeds.h"  
  
configuration RadioCountToLedsAppC {}  
implementation {  
    components MainC, RadioCountToLedsC as App, LedsC;  
    components new AMSenderC(AM_RADIO_COUNT_MSG);  
    components new AMReceiverC(AM_RADIO_COUNT_MSG);  
    components new TimerMilliC();  
    components ActiveMessageC;  
  
    App.Boot -> MainC.Boot;  
    App.Receive -> AMReceiverC;  
    App.AMSend -> AMSenderC;  
    App.AMControl -> ActiveMessageC;  
    App.Leds -> LedsC;  
    App.MilliTimer -> TimerMilliC;  
    App.Packet -> AMSenderC;  
}
```

## RadioCountToLeds



- RadioCountToLedsC.nc

```
#include "Timer.h"
#include "RadioCountToLeds.h"

module RadioCountToLedsC {
  uses {
    interface Leds;
    interface Boot;
    interface Receive;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface SplitControl as AMControl;
    interface Packet;
  }
}
implementation {
  message_t packet;
  bool locked;
  uint16_t counter = 0;
}
```

## RadioCountToLeds



- RadioCountToLedsC.nc

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

event void AMControl.startDone(error_t err) {
  if (err == SUCCESS) {
    call MilliTimer.startPeriodic(250);
  }
  else {
    call AMControl.start();
  }
}

event void AMControl.stopDone(error_t err) {
  // do nothing
}
```

## RadioCountToLeds



- RadioCountToLedsC.nc

```
event void MilliTimer.fired() {
    counter++;
    if (locked) return;
    else
    {
        radio_count_msg_t* rcm =
            (radio_count_msg_t*) call Packet.getPayload(&packet, NULL);

        if (call Packet.maxPayloadLength() < sizeof(radio_count_msg_t))
            return;

        rcm->counter = counter;

        if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
            sizeof(radio_count_msg_t)) == SUCCESS)
            locked = TRUE;
    }
}
```

## RadioCountToLeds



- RadioCountToLedsC.nc

```
event void AMSend.sendDone(message_t* bufPtr, error_t error) {
    if (&packet == bufPtr) {locked = FALSE;}
}
}
```

## RadioCountToLeds

- RadioCountToLedsC.nc

```
event message_t*
Receive.receive(message_t* bufPtr, void* payload, uint8_t len){
    if (len != sizeof(radio_count_msg_t)) {return bufPtr;}
    else{
        radio_count_msg_t* rcm = (radio_count_msg_t*)payload;

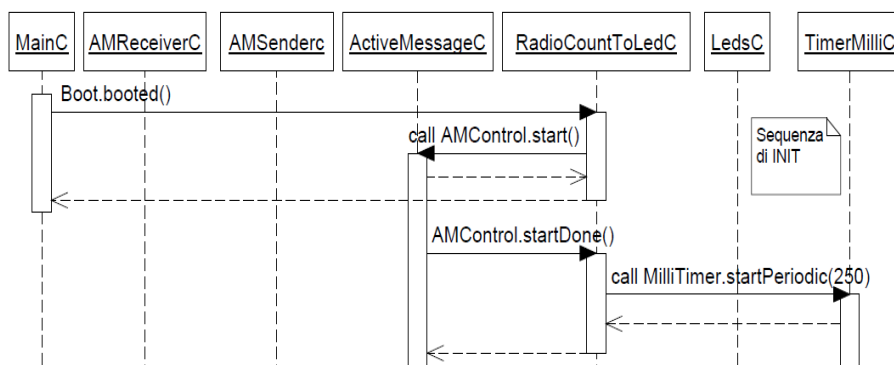
        if (rcm->counter & 0x1) call Leds.led0On();
        else call Leds.led0Off();

        if (rcm->counter & 0x2) call Leds.led1On();
        else call Leds.led1Off();

        if (rcm->counter & 0x4) call Leds.led2On();
        else call Leds.led2Off();
    }
    return bufPtr;
}
```

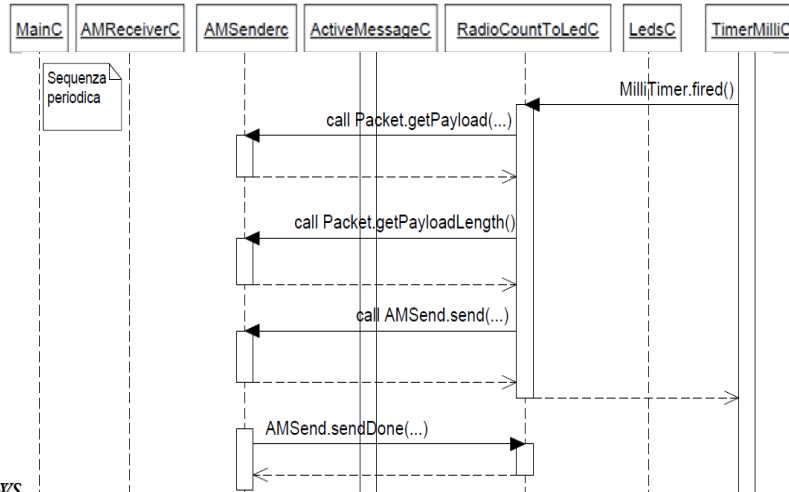
## RadioCountToLeds

- Sequence diagram: INIT



## RadioCountToLeds

- Sequence diagram: attività periodica

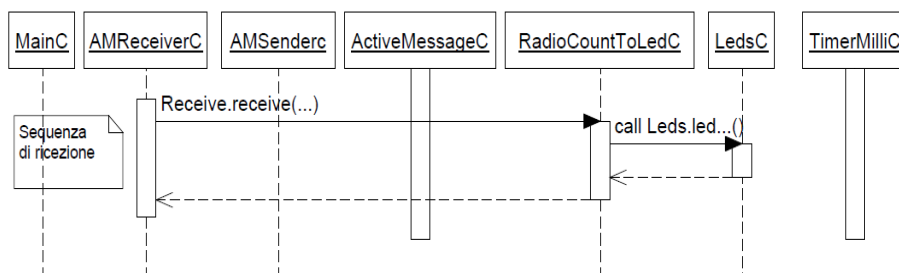


DEWS

113

## RadioCountToLeds

- Sequence diagram: ricezione



DEWS

114

## Esempi nesC/TinyOS

RadioSenseToLeds (TinyOS 2.x)

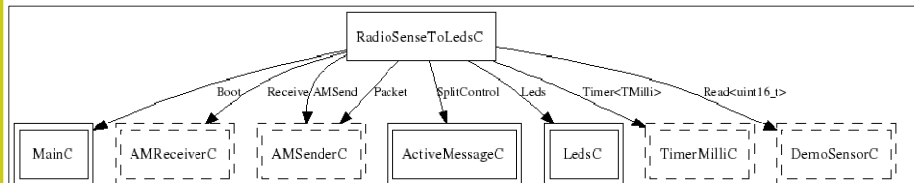
### RadioSenseToLeds



- Campiona periodicamente (4Hz) un sensore e spedisce il valore letto via radio in broadcast
  - I nodi che ricevono visualizzano i tre LSB sui led
- Componenti applicativi
  - *RadioSenseToLedsAppC (Configuration)*
  - *RadioSenseToLedsC (Module)*
- Componenti di sistema
  - *MainC, LedsC, TimerMilliC, DemoSensorC*
  - *AMReceiverC, AMSenderC, ActiveMessageC*

## RadioSenseToLeds

- RadioSenseToLedsAppC: grafo dei componenti



## RadioSenseToLeds

- RadioSenseToLeds.h

```
typedef nx_struct radio_sense_msg {
    nx_uint16_t error;
    nx_uint16_t data;
} radio_sense_msg_t;

enum {
    AM_RADIO_SENSE_MSG = 7,
};
```

## RadioSenseToLeds



- RadioSenseToLedsAppC.nc

```
#include "RadioSenseToLeds.h"
configuration RadioSenseToLedsAppC {}
implementation {
    components MainC, RadioSenseToLedsC as App, LedsC, new DemoSensorC();
    components ActiveMessageC;
    components new AMSenderC(AM_RADIO_SENSE_MSG);
    components new AMReceiverC(AM_RADIO_SENSE_MSG);
    components new TimerMilliC();

    App.Boot -> MainC.Boot;
    App.Receive -> AMReceiverC;
    App.AMSend -> AMSenderC;
    App.RadioControl -> ActiveMessageC;
    App.Leds -> LedsC;
    App.MilliTimer -> TimerMilliC;
    App.Packet -> AMSenderC;
    App.Read -> DemoSensorC;
}
```

## RadioSenseToLeds



- RadioSenseToLedsC.nc

```
#include "Timer.h"
#include "RadioSenseToLeds.h"
module RadioSenseToLedsC {
    uses {
        interface Leds;
        interface Boot;
        interface Receive;
        interface AMSend;
        interface Timer<TMilli> as MilliTimer;
        interface Packet;
        interface Read<uint16_t>;
        interface SplitControl as RadioControl;
    }
}
implementation {
    message_t packet;
    bool locked = FALSE;
```

## RadioSenseToLeds



- RadioSenseToLedsC.nc

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

event void RadioControl.startDone(error_t err) {
    if (err == SUCCESS) {
        call MilliTimer.startPeriodic(250);
    }
}

event void RadioControl.stopDone(error_t err) {}
```

## RadioSenseToLeds



- RadioSenseToLedsC.nc

```
event void MilliTimer.fired() {call Read.read();}

event void Read.readDone(error_t result, uint16_t data) {
    if (locked) return;
    else
    { radio_sense_msg_t* rsm;

        rsm = (radio_sense_msg_t*)call Packet.getPayload(&packet, NULL);
        if (call Packet.maxPayloadLength() < sizeof(radio_sense_msg_t))
            return;

        rsm->error = result;
        rsm->data = data;
        if (call AMSend.send(AM_BROADCAST_ADDR, &packet,
            sizeof(radio_sense_msg_t)) == SUCCESS) locked = TRUE;
    }
}
```

## RadioSenseToLeds



- RadioSenseToLedsC.nc

```
event void AMSend.sendDone(message_t* bufPtr, error_t error) {  
    if (&packet == bufPtr) {locked = FALSE;}  
}  
}
```

## RadioSenseToLeds



- RadioSenseToLedsC.nc

```
event message_t*  
Receive.receive(message_t* bufPtr, void* payload, uint8_t len){  
    if (len != sizeof(radio_count_msg_t)) {return bufPtr;}  
    else{  
        radio_count_msg_t* rcm = (radio_count_msg_t*)payload;  
  
        if (rcm->counter & 0x0001) call Leds.led0On();  
        else call Leds.led0Off();  
  
        if (rcm->counter & 0x0002) call Leds.led1On();  
        else call Leds.led1Off();  
  
        if (rcm->counter & 0x0004) call Leds.led2On();  
        else call Leds.led2Off();  
    }  
    return bufPtr;  
}  
}
```

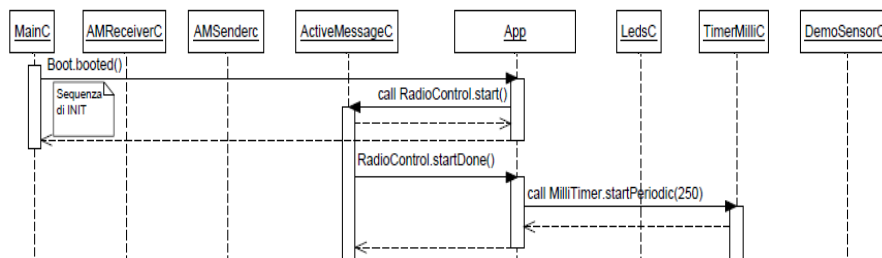
## RadioSenseToLeds

- DemoSensorC.nc

```
generic configuration DemoSensorC()
{
    provides interface Read<uint16_t>;
}
implementation
{
    components new VoltageC() as DemoChannel;
    Read = DemoChannel;
}
```

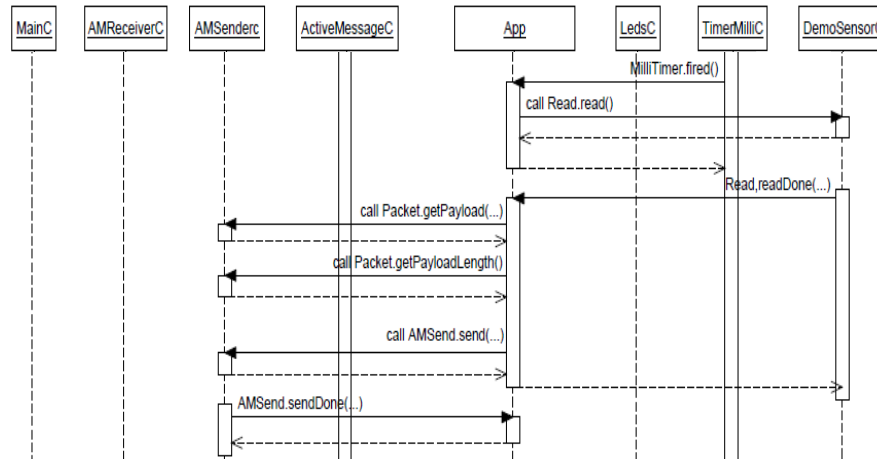
## RadioSenseToLeds

- Sequence diagram: INIT



## RadioSenseToLeds

- Sequence diagram: attività periodica

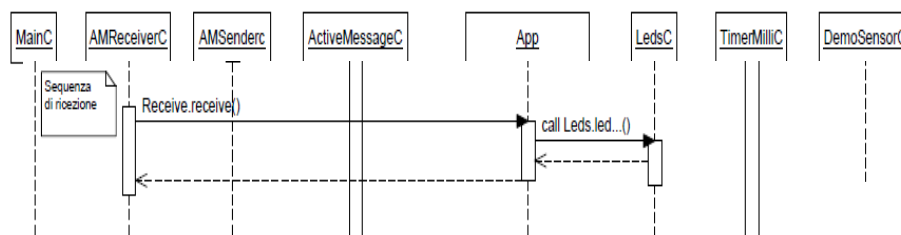


DEWS

127

## RadioSenseToLeds

- Sequence diagram: ricezione



DEWS

128

## Esempi nesC/TinyOS

Oscilloscope (TinyOS 2.x)

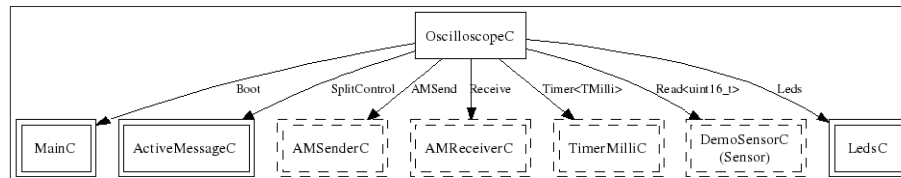
### Oscilloscope



- Campiona periodicamente (4Hz) un sensore e spedisce un messaggio contenente 10 valori
  - I dati ricevuti dal nodo sink sono visualizzati su PC
- Componenti applicativi
  - *OscilloscopeAppC (Configuration)*, *OscilloscopeC (Module)*
    - *BaseStation*
- Componenti di sistema
  - *MainC*, *LedsC*, *TimerMilliC*, *DemoSensorC*
  - *AMReceiverC*, *AMSenderC*, *ActiveMessageC*

## Oscilloscope

- OscilloscopeAppC: grafo dei componenti



## Oscilloscope

- Oscilloscope.h

```

enum
{
    NREADINGS = 10,
    DEFAULT_INTERVAL = 256,
    AM_OSCILLOSCOPE = 0x93
};

typedef nx_struct oscilloscope
{
    nx_uint16_t version;          /* Version of the interval. */
    nx_uint16_t interval;        /* Sampling period. */
    nx_uint16_t id;              /* id of sending mote. */
    nx_uint16_t count;           /* The readings are samples count *
                                NREADINGS onwards */
    nx_uint16_t readings[NREADINGS];
} oscilloscope_t;
  
```

# Oscilloscope



- OscilloscopeAppC.nc

```
#include "Oscilloscope.h"

configuration OscilloscopeAppC { }
implementation {
    components OscilloscopeC, MainC, ActiveMessageC, LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE),
    new AMReceiverC(AM_OSCILLOSCOPE);

    OscilloscopeC.Boot -> MainC;
    OscilloscopeC.RadioControl -> ActiveMessageC;
    OscilloscopeC.AMSend -> AMSenderC;
    OscilloscopeC.Receive -> AMReceiverC;
    OscilloscopeC.Timer -> TimerMilliC;
    OscilloscopeC.Read -> Sensor;
    OscilloscopeC.Leds -> LedsC;
}
```

# Oscilloscope



- OscilloscopeC.nc

```
#include "Timer.h"
#include "Oscilloscope.h"
module OscilloscopeC{
    uses {
        interface Boot;
        interface SplitControl as RadioControl;
        interface AMSend;
        interface Receive;
        interface Timer<TMilli>;
        interface Read<uint16_t>;
        interface Leds;}
    }
    implementation{
        message_t sendBuf;
        bool sendBusy;
        oscilloscope_t local;
        uint8_t reading;
        bool suppressCountChange;
```

# Oscilloscope



- OscilloscopeC.nc

```
// Utility C-style functions
void report_problem() { call Leds.led0Toggle(); }
void report_sent() { call Leds.led1Toggle(); }
void report_received() { call Leds.led2Toggle(); }
void startTimer()
{call Timer.startPeriodic(local.interval); reading = 0;}

event void Boot.booted() {
  local.interval = DEFAULT_INTERVAL;
  local.id = TOS_NODE_ID;
  if (call RadioControl.start() != SUCCESS)
    report_problem();
}

event void RadioControl.startDone(error_t error) {
  startTimer();
}
event void RadioControl.stopDone(error_t error) {}
```

# Oscilloscope



- OscilloscopeC.nc

```
event void Timer.fired(){
  if (reading == NREADINGS)
  {
    if (!sendBusy&&sizeof local <= call AMSend.maxPayloadLength())
    {
      memcpy(call AMSend.getPayload(&sendBuf,
        sizeof(local)), &local, sizeof local);
      if (call AMSend.send(AM_BROADCAST_ADDR, &sendBuf,
        sizeof local) == SUCCESS) sendBusy = TRUE;
    }

    if (!sendBusy) report_problem();
    reading = 0;
    if (!suppressCountChange) local.count++;
    suppressCountChange = FALSE;
  }
  if (call Read.read() != SUCCESS) report_problem();
}
```

# Oscilloscope



- OscilloscopeC.nc

```
event void Read.readDone(error_t result, uint16_t data)
{
    if (result != SUCCESS)
    {
        data = 0xffff;
        report_problem();
    }
    local.readings[reading++] = data;
}
}
```

# Oscilloscope



- OscilloscopeC.nc

```
event void AMSend.sendDone(message_t* msg, error_t error)
{ if (error == SUCCESS) report_sent();
  else report_problem();
  sendBusy = FALSE; }

event message_t*
Receive.receive(message_t* msg, void* payload, uint8_t len)
{ oscilloscope_t *omsg = payload;
  report_received();

  if (omsg->version > local.version)
  { local.version = omsg->version; local.interval = omsg->interval;
    startTimer(); }
  if (omsg->count > local.count)
  { local.count = omsg->count; suppressCountChange = TRUE; }
  return msg;
}
```

## Oscilloscope



- BaseStation
  - Applicazione da installare sul *nodo sink* collegato via seriale ad un PC
    - Essa inoltra via radio i dati ricevuti dalla seriale e inoltra su seriale (verso il PC) i pacchetti ricevuti dalla radio
      - I dati scambiati devono rispettare la struttura dei pacchetti AM

## Conclusioni

# Conclusioni



## *The Internet of Things*

