Sapienza Università di Roma

Wireless Systems Laboratory

October 14, 2013

A. Cammarano, A.Capossele, D. Spenza

Contacts

Cammarano: cammarano@di.uniroma1.it
Capossele: capossele@di.uniroma1.it
Spenza: spenza@di.uniroma1.it
Google Group:
http://groups.google.com/d/forum/sistemiwireless2013-di-uniroma1

Tel: 06-49918430
Room: 333

Slides: wwwusers.di.uniroma1.it/~spenza/lab2013.html

Outline

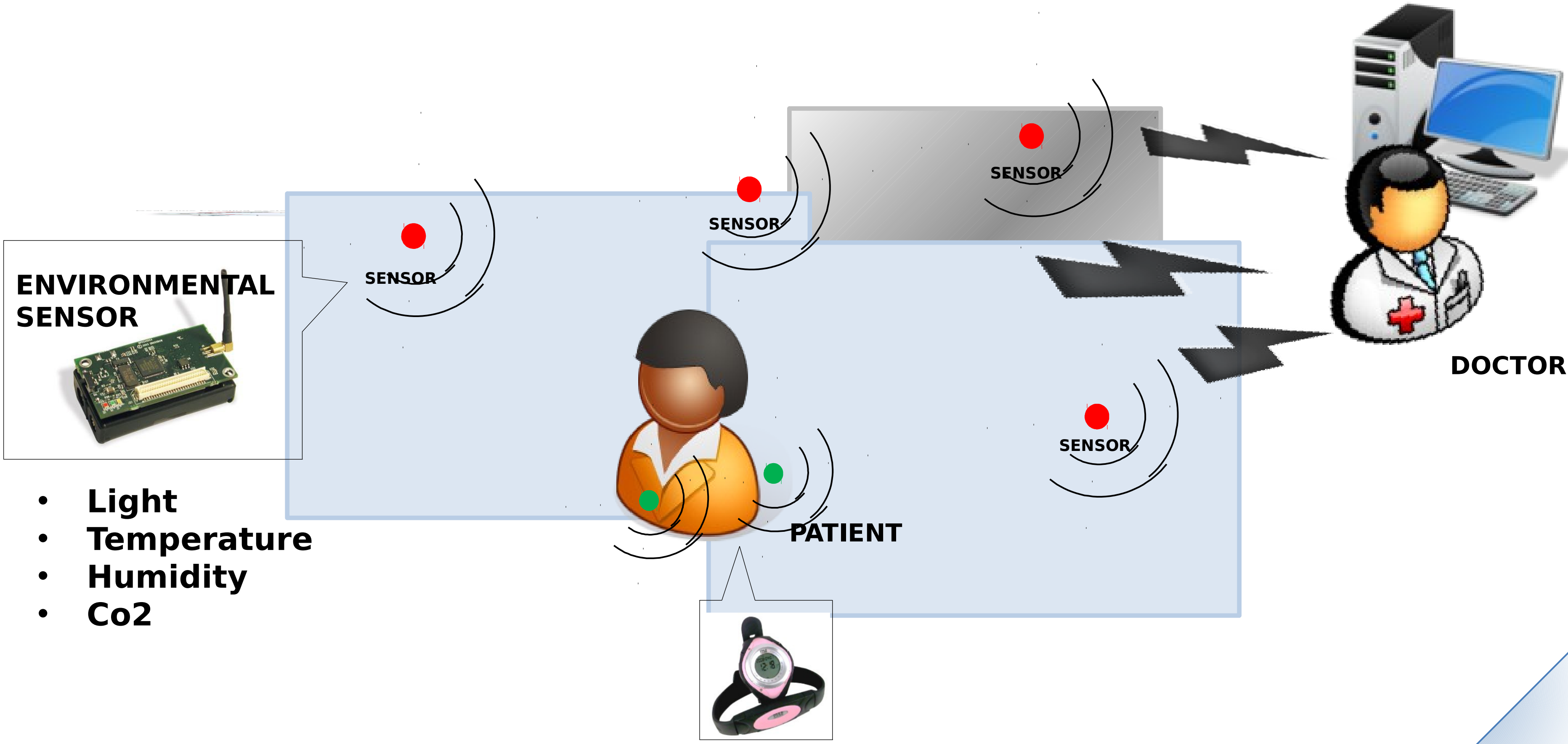
- SENSES lab
- WSN: introduction, examples
- NesC
- TinyOS
- A simple application: Blink

Wireless sensor networks (WSN) are nowadays being deployed in a large number of application domains

- military environments and perimeter sensing weather and ambient control
- industrial applications
- power grids
- health care
- Security – Harvesting – Cognitive Network

CHIRON

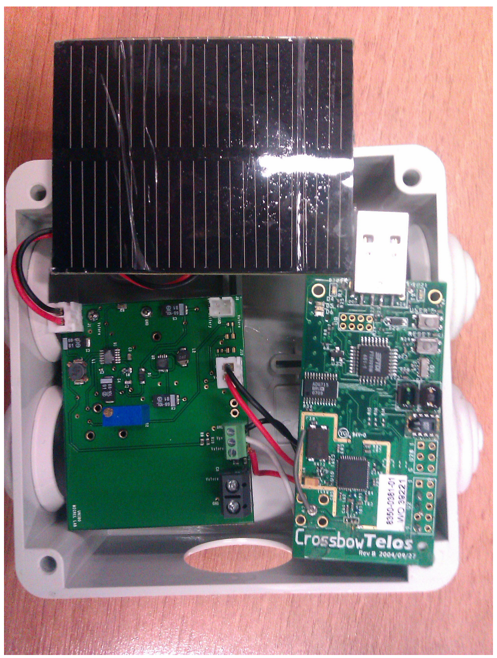
Correlation between patient's condition and environment

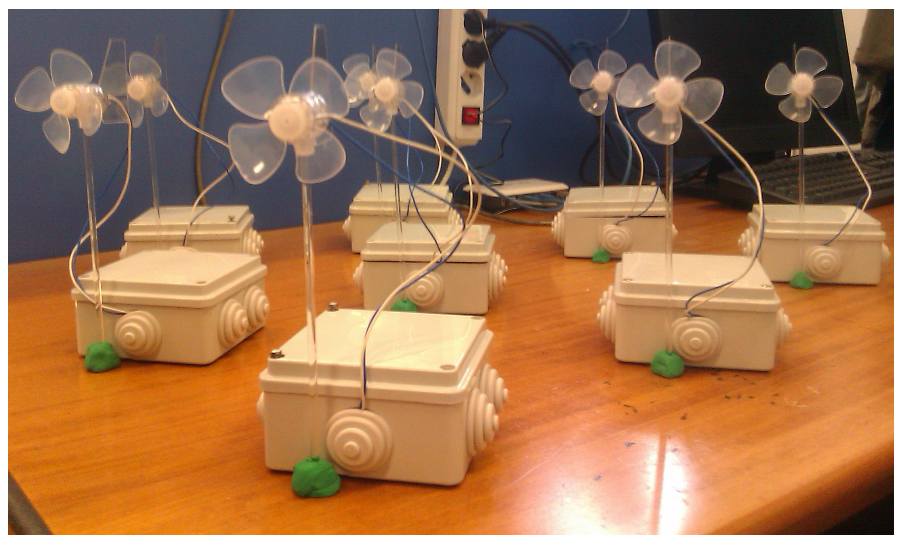


- Light
- Temperature
- Humidity
- Co2

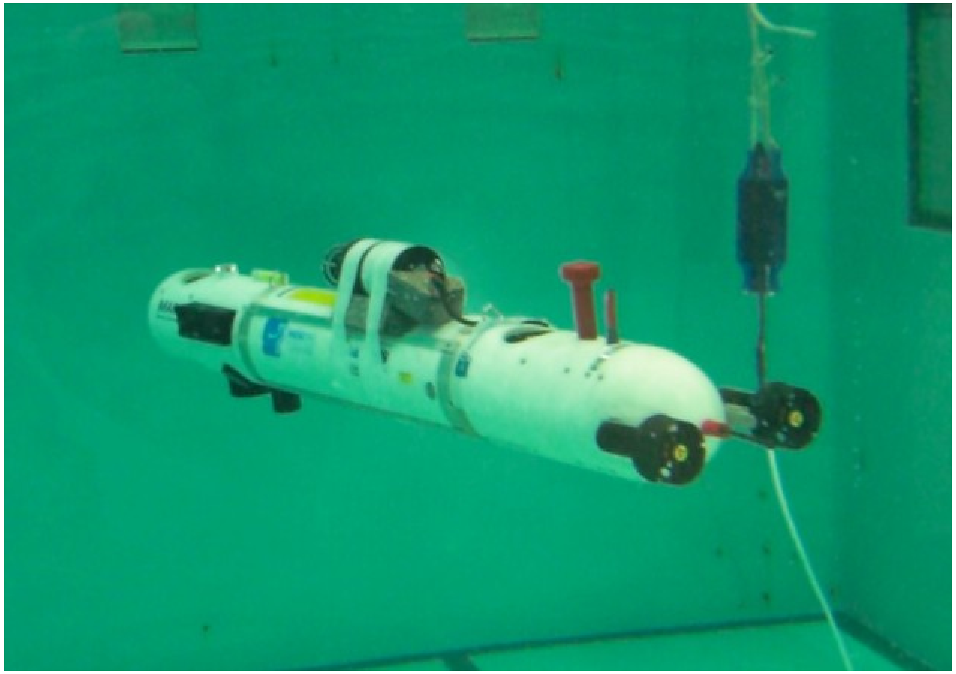
GENESI

Green Wireless Sensor Networks
Energy Harvesting to extend
WSN's life-time.





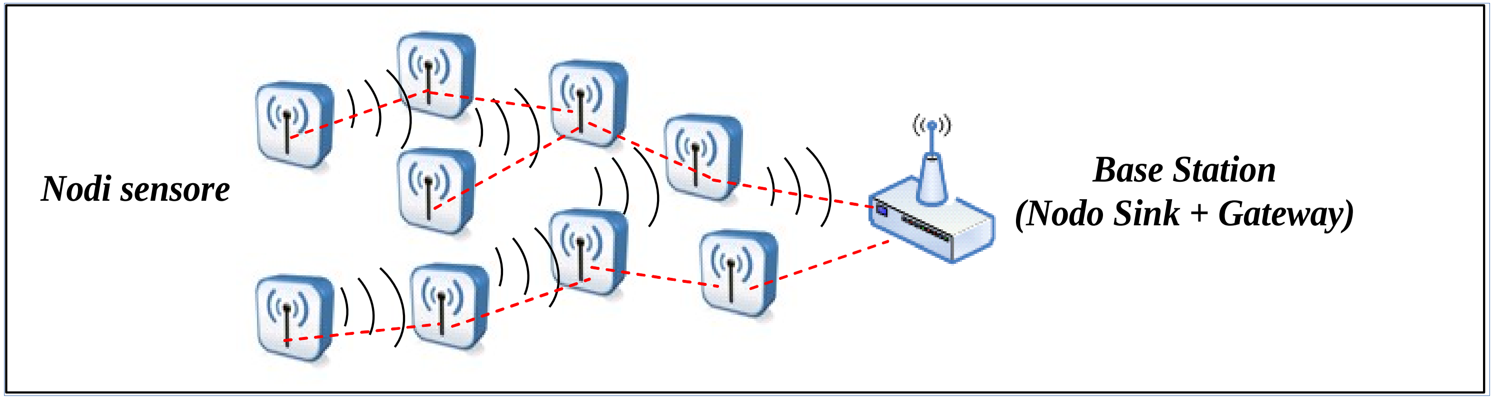
Underwater sensor networks



Acoustic communication

- Monitoring: oil, gas, CO_2;
- Natural disaster prevention;
- Chemical composition of ocean floor;
- ...

WSN architecture



Through the sensory component of a node, physical qualities of the areas where the network is deployed can be measured.
WSNs data are generated at the sensor nodes and are forwarded to a Base Station (Sink)

- Sensor node (*node*, *mote*) and *Base Station*
- *Wireless communication (multi-hop)*

WSN

- sensor
  - A transducer
  - converts physical phenomenon e.g. heat, light, motion, vibration, and sound into electrical signals
- sensor node
  - basic unit in sensor network
  - contains on-board sensors, processor, memory, transceiver, and power supply
- sensor network
  - consists of a large number of sensor nodes
  - nodes deployed either inside or very close to the sensed phenomenon

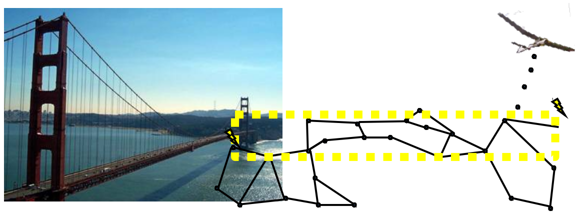
Factors Influencing WSN Design

- Fault tolerance
- Scalability
- Production costs
- Hardware constraints
- Sensor network topology
- Environment
- Transmission media
- Power Consumption
  - Sensing
  - Communication
  - Data processing

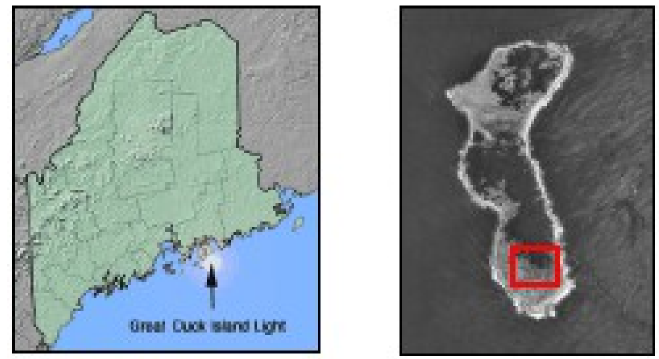
Applications

Applications:
- Military
- Environmental
- Health-care
- Home-automation
- Industrial
- Civil
- ...

More applications



Bridge monitoring



Great Duck Island

Study on Petrel (birds)

Patient monitoring

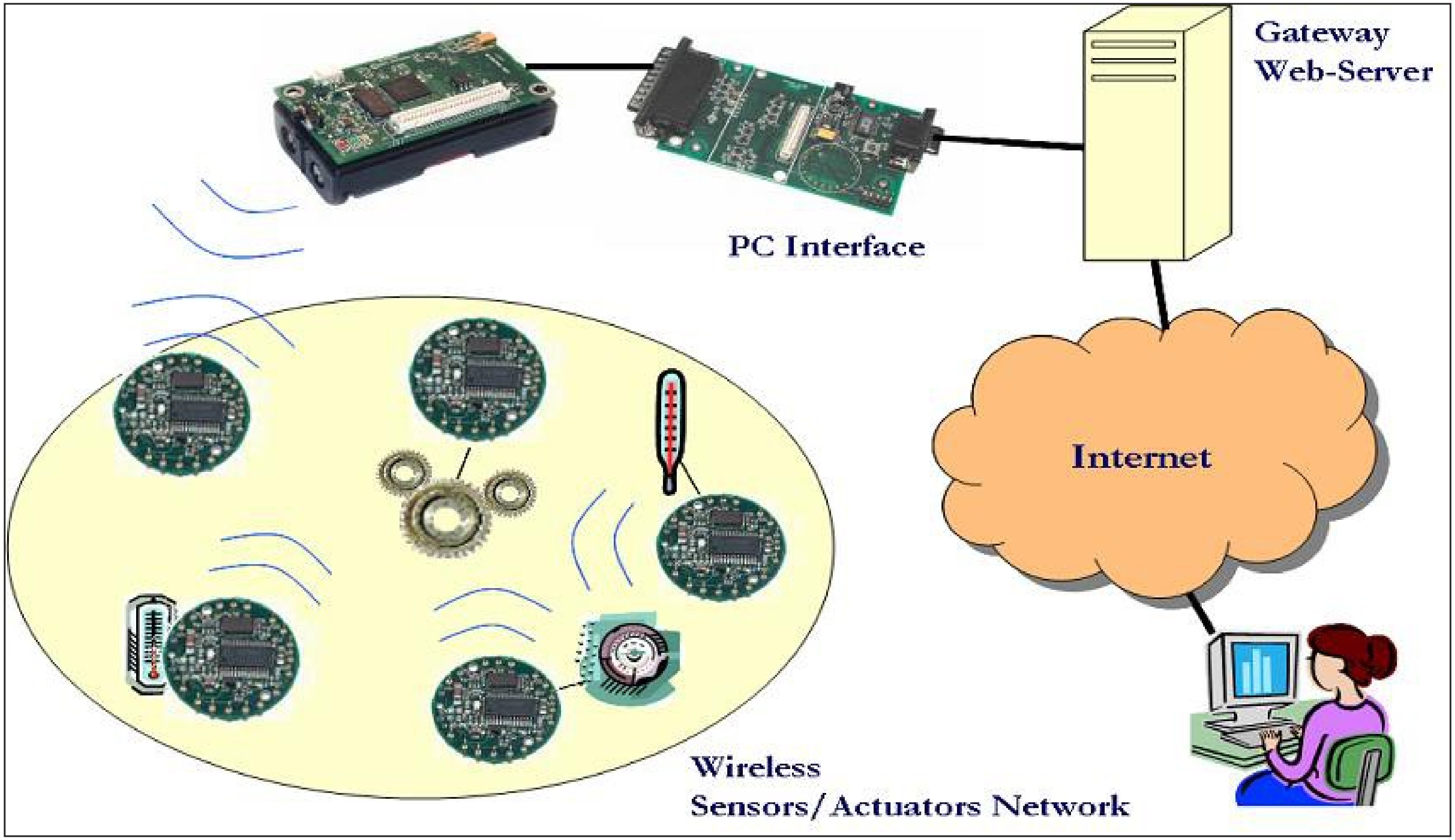
Precision farming

More applications

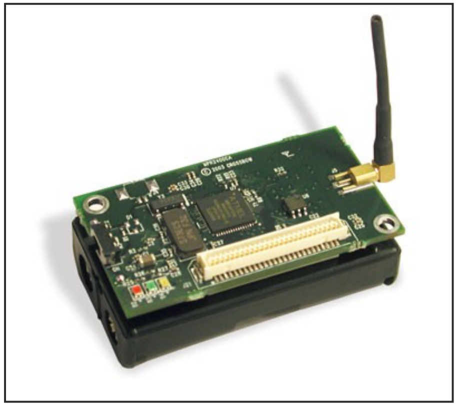
Other Commercial Applications

- Environmental control in office buildings (estimated energy savings $55 billion per year!)
- Interactive museums
- Detecting and monitoring car thefts
- Managing inventory control
- Vehicle tracking and detection

Hardware


Gateway
Web-Server
PC Interface
Internet
Wireless
Sensors/Actuators Network

Mica2



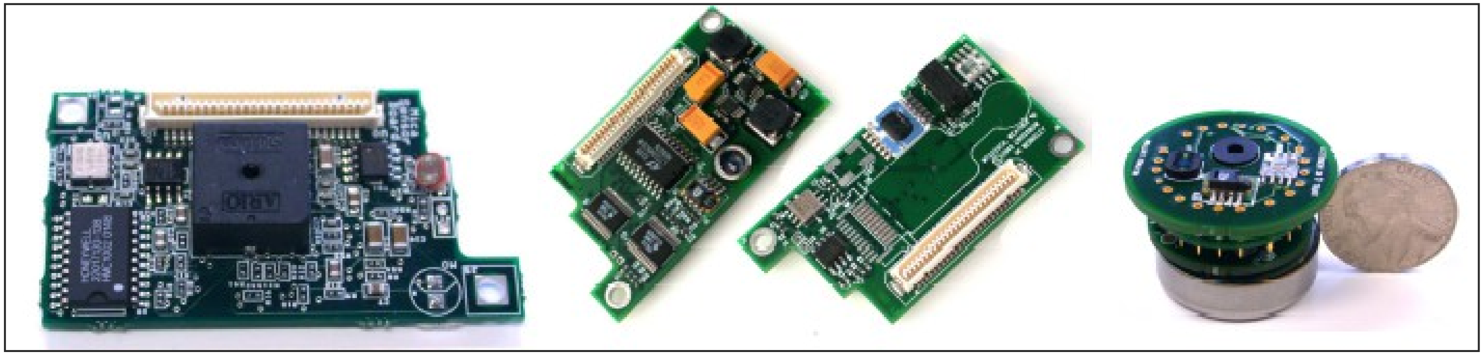
- CPU: microcontrollor Atmel ATmega128L
  - MPU: 8-bit RISC (0-8 MHz)
  - Memory
    - ROM: 128K Bytes Flash
    - RAM: 4K Bytes SRAM
  - ADC, UART, GPIO, I2C, SPI, Timer
- Communication: *Transceiver* Chipcon CC1000
  - 868/915 MHz, 38.4 kbps, range 30-100 m)
- Local storage: Flash 512 KB

TelosB

- CPU: microcontrollor TI MSP430
  - MPU: 16-bit RISC (0-8 MHz)
  - Memory
    - ROM: 48K Bytes Flash
    - RAM: 10K Bytes SRAM
  - ADC, UART, GPIO, I2C, SPI, Timer
- Communication: *Transceiver* Chipcon CC2420
  - IEEE 802.15.4 (2,4 GHz, 250 kbps, range 20-100 m)
- Local Storage: Flash 1024 KB

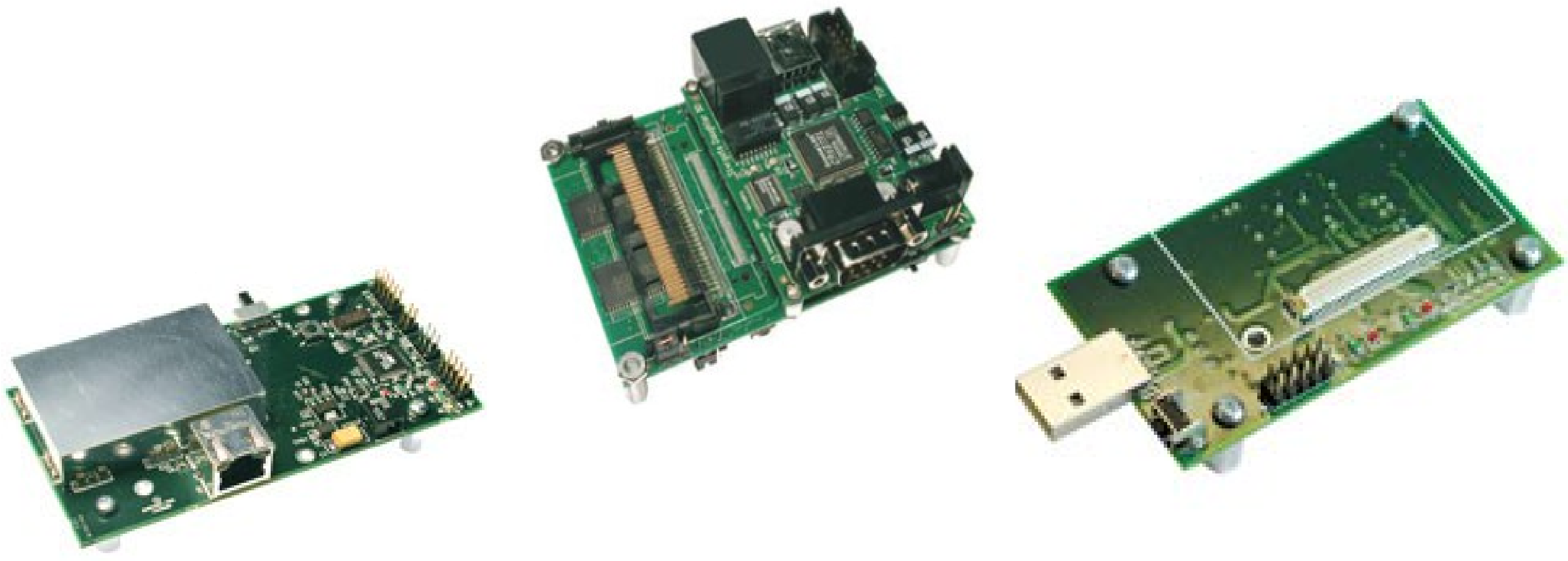
Sensor Board

- Several kinds of “sensor”
  - Light, temperature, pressure, humidity
  - Accelerometer, magnetometer, distance
  - Microphones, videocameras, GPS



Base Station

- *Wired link* PC-node (*wireless* with other nodes)
  - Parallel, serial (*MIB 510/520*), ethernet



TinyOS

- TinyOS began as a collaboration between University of California, Berkeley and Intel Research.
- It is a free open source operating system designed for wireless sensor networks.
- It is an embedded operating system written in **NesC** (network embedded system C).
- It features a **component based** architecture.

TinyOS - nesC

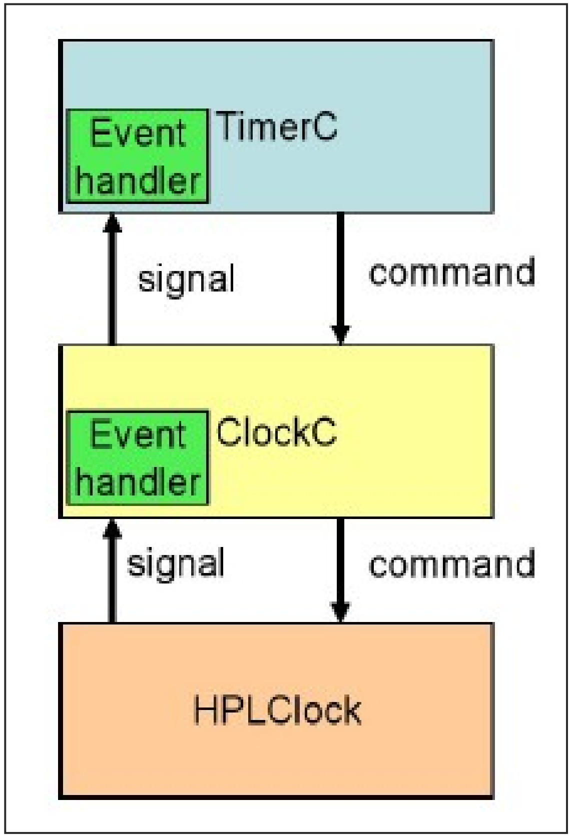
- Separation construction/composition
- Construction of Modules
- Modules implementation similar to C coding
  - Programs are built out of components
  - Each component specifies an interface
  - Interfaces are “hooks” for wiring components
- Composition of Configurations
  - Components are statically wired together
  - Increases programming efficiency (code reuse) and runtime efficiency (static defs.)

Component Model

- Components should use and provide bidirectional interfaces.
-
- Components should call and implement commands and signal and handle events.
-
- Components must handle events of used interfaces and also provide interfaces that must implement commands.

Component Model: Hierarchy

- Commands
  - Flow downwards
  - Non Blocking requests
  - Control returns to caller
- Events
  - Flow upwards
  - Post task, signal higher level events, call lower level cmds
  - Control returns to signaler
- To avoid cycles
  - Events can call commands
  - Commands can NOT signal events



Example – Component: module

```
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() {
    …
  }
}
```

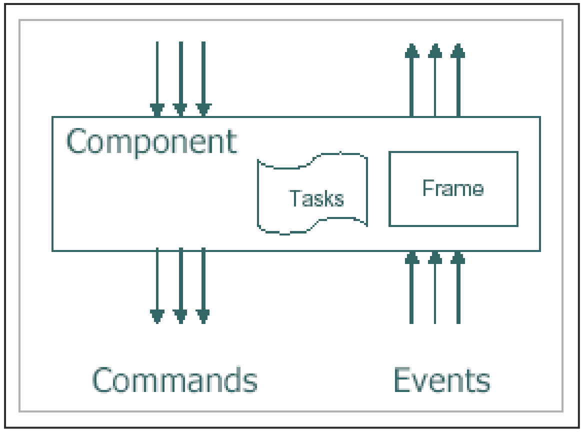
Example – Component: configuration

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

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

TinyOS - nesC

- **Tasks** enable components to perform general-purpose "background" processing in an application
  - Event
    - High priority
  - Task
    - Low priority



TinyOS guarantees that task will *eventually* run.

TinyOS - nesC

When you are developing an application for TinyOS, keep in mind:

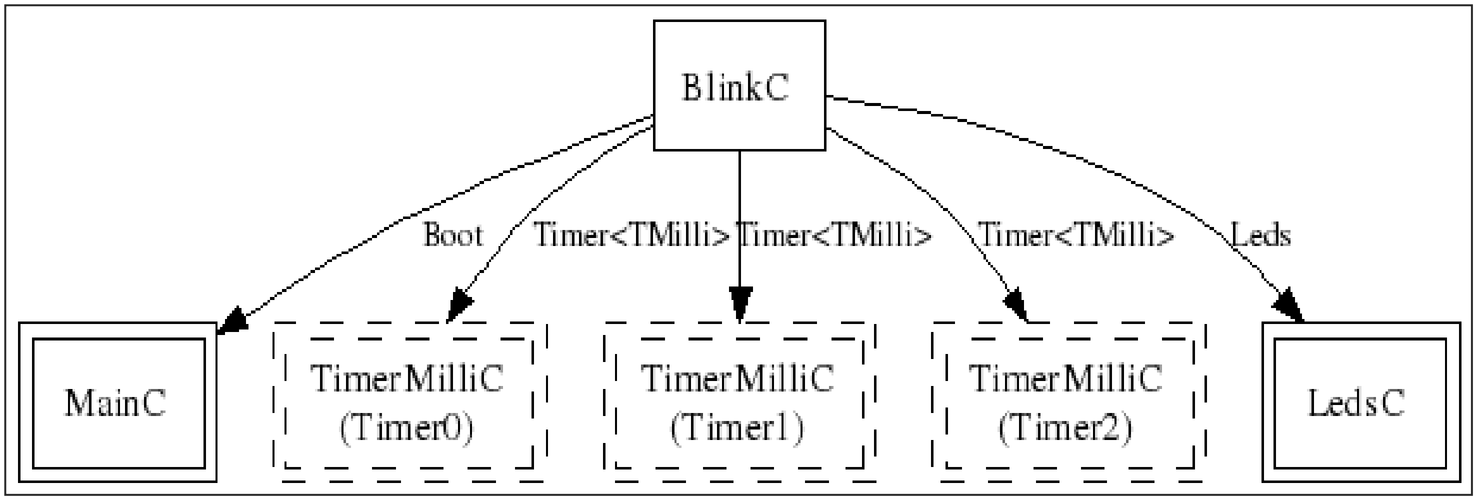
Hurry Up and Sleep!!!

- In order to save battery life a node should be in the *sleep* state as much as possible
- When an *event* wakes up a node, the node should do something and then return in the sleep state.
    - *Interrupt-driven & Split-phase*

Example - Blink

The application displays a counter on the three mote LEDs
- Leds turn on and off at 1Hz, 2Hz, and 4Hz
- Application components:
  - *BlinkAppC (Configuration)*
  - *BlinkC (Module)*
- System components:
  - *MainC, LedsC, TimerMilliC*

BlinkAppC components graph:



	Singleton	Generic
Module		
Configuration		

Example - 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;
}
```

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
{
```

Example - Blink

BlinkC.nc

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

Example - 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();
  }
}
```

Example – Blink counter

BlinkC.nc

```
uint8_t counter = 0;

event void Boot.booted()
{
  call Timer0.startPeriodic( 1024 );
}
```

	8 bits	16 bits	32 bits	64 bits
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

Example – Blink counter

BlinkC.nc

```
event void Timer0.fired()
{
    counter++;
    if (counter & 0x1) {
      call Leds.led0On();
    }
    else {
      call Leds.led0Off();
    }
    if (counter & 0x2) {
      call Leds.led1On();
    }
    else {
      call Leds.led1Off();
    }
    if (counter & 0x4) {
      call Leds.led2On();
    }
    else {
      call Leds.led2Off();
    }
}
```