Internet of Things Laboratory

November 9th 2015

P. Gjanci, G. Koutsandria, D. Spenza

Contacts

- Gjanci: gjanci@di.uniroma1.it
- Koutsandria: koutsandria@di.uniroma1.it
- Spenza: spenza@di.uniroma1.it
  - Tel: 06-49918430
  - Room: 333
  - Slides: www.dsi.uniroma1.it/~spenza/

- SENSES lab
  - http://senseslab.di.uniroma1.it

Lessons Schedule

09.11.2015
8:30-12:00
Lesson 1

10.11.2015
17:15-19:30
Tutoring session

13.11.2015
14:00-17:30
Lesson 2

23.11.2015
08:30-12:00
Lesson 3

24.11.2015
17:15-19:30
Tutoring/project?

27.11.2015
14:00-17:30
Final project

Mailing list

- Subscribe to the course mailing list: iot-laboratory-diuniroma1@googlegroups.com

- Slides and supporting material will be made available online at: http://wwwusers.di.uniroma1.it/~spenza/lab2015.html

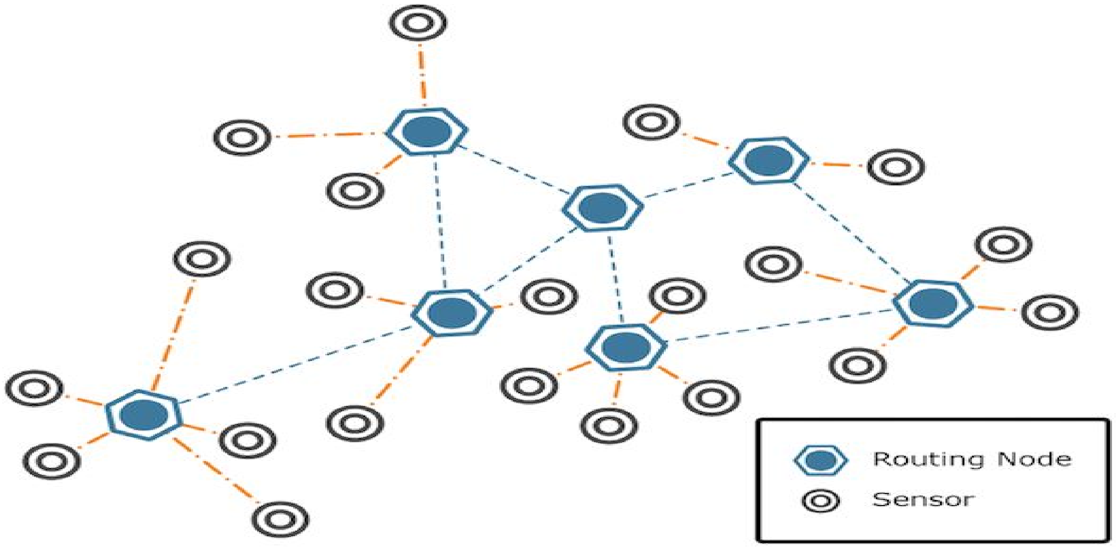
- All lectures will be active learning sessions with lab exercises.

Overview

- Introduction on Wireless Sensor Networks
- WSNs Projects of the SENSES lab
- Applications on Wireless Sensor Networks
- TinyOS introduction
- NesC programming language
- A simple application: Blink

Introduction: Wireless Sensor Networks

- *Network of small wireless sensor devices (nodes/motes), deployed in an ad-hoc fashion to cooperate on sensing a physical or environmental phenomenon.*



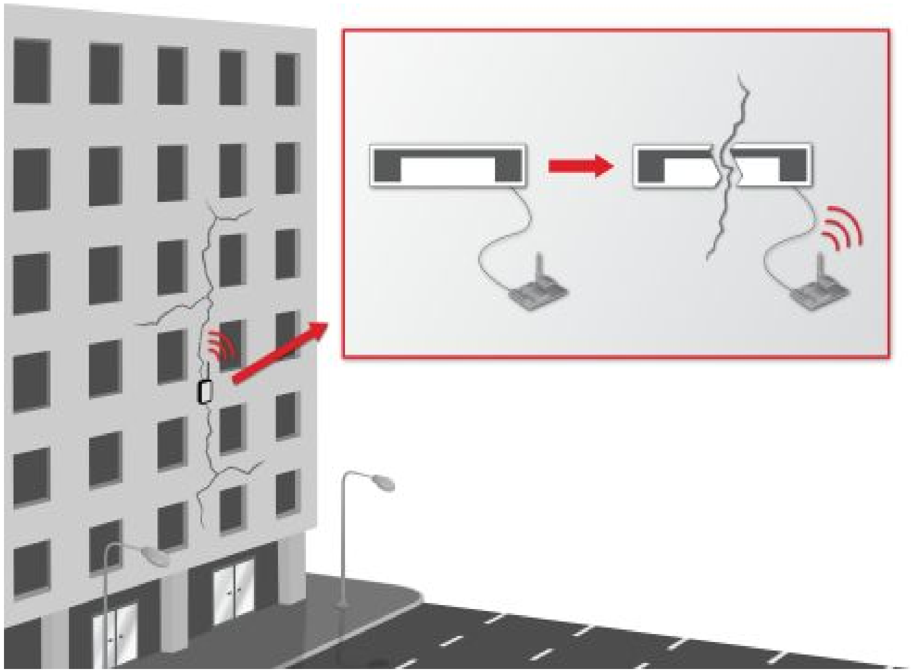
- Wireless communication medium
- Traffic is forwarded through several hops from source to sink node
- Power limited

Application Scenarios

- Environmental monitoring
  - fire/flood detection
- Monitoring of cultural heritage
  - “health” status of artworks
- Structural monitoring
  - integrity
  - life signs
- Medical
  - patient’s health status
- Military
  - surveillance
- Home automation
- Etc..

Structural Health Monitoring

- Construction sites: inclinometers, pressure, displacement, …
- Bridges, buildings: vibrating-wire strain gauges, displacement, ...



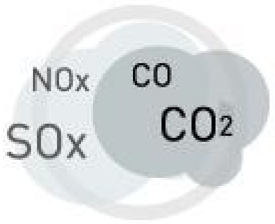
Cultural Heritage Preservation

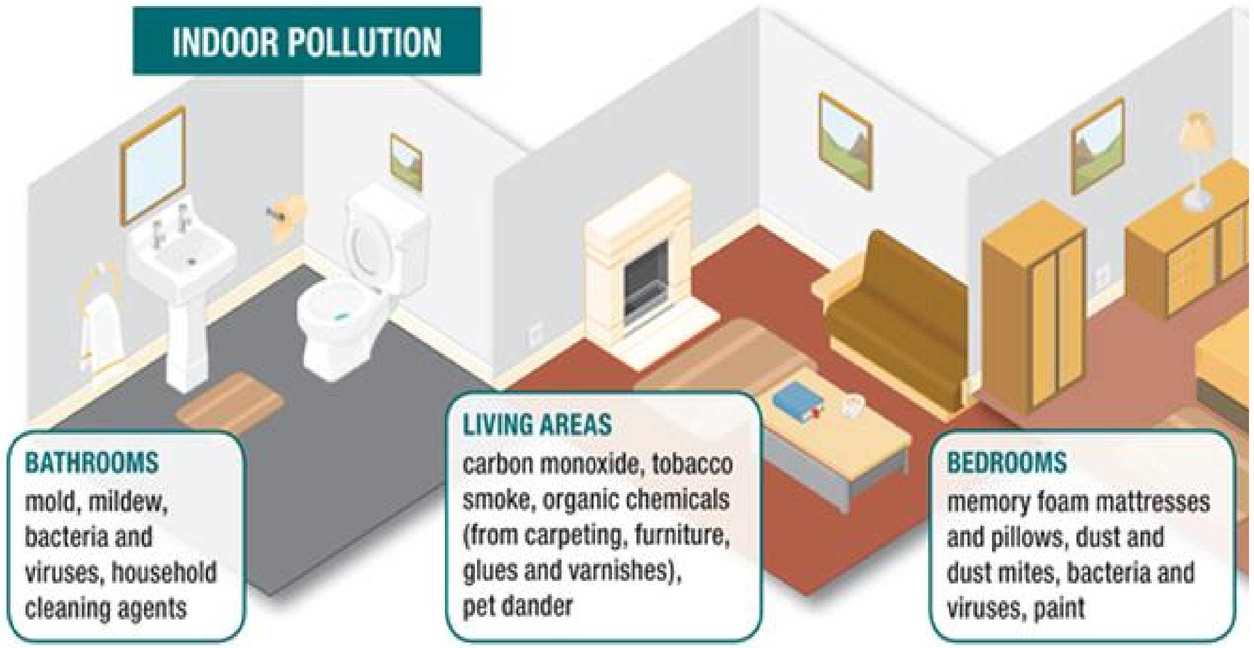
- Indoor/outdoor preservation: temperature, humidity, dust…
- Artworks monitoring during transportation and packing

CULTURAL
HERITAGE
A CHALLENGE
FOR EUROPE

Environmental Monitoring

- Outdoor air pollution: carbon monoxide, temperature, humidity,...
- Indoor air quality: CO2, carbon monoxide, dust level, humidity...
- Gas detection: methane, carbon monoxide,...

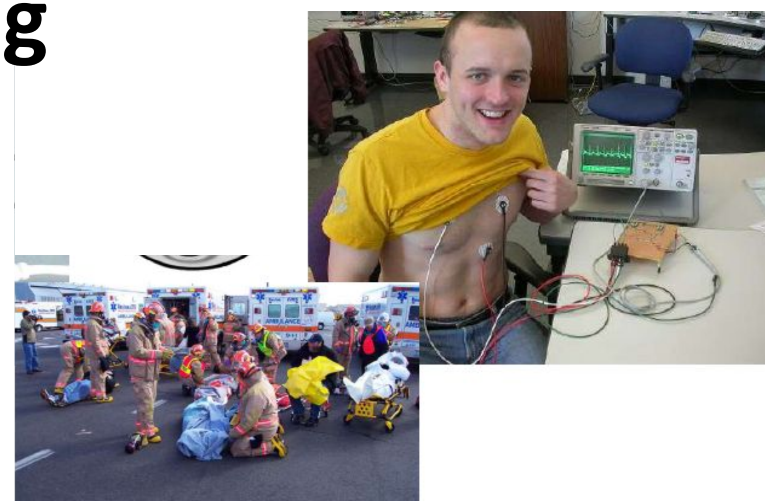




Habitat Monitoring

- **Great Duck Island Project**
  - 150 Sensing nodes
  - Temperature, pressure, humidity
  - Data available on the Internet through a satellite link

Healthcare, Assisted Living

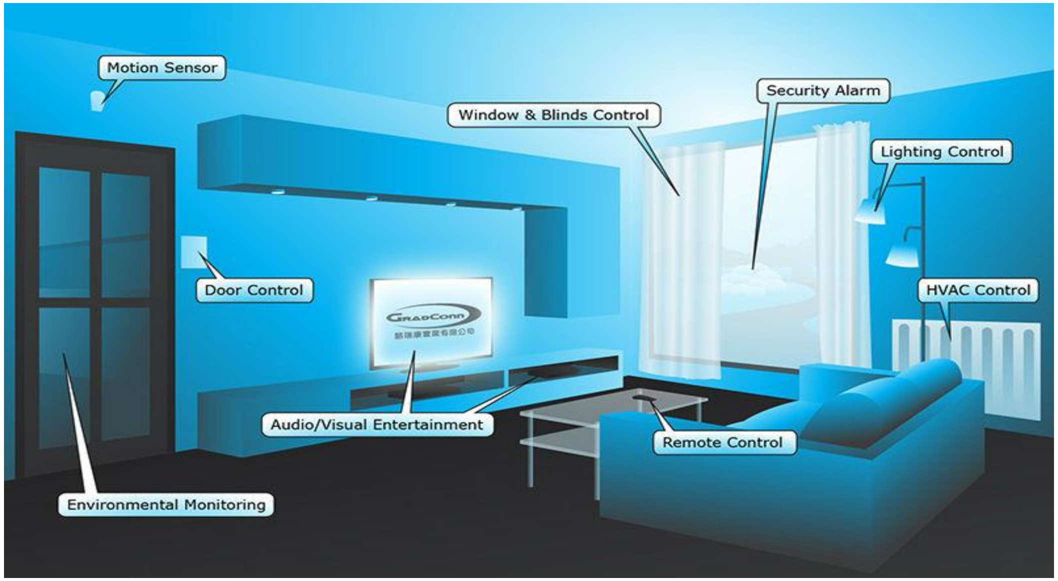


- Vital sign monitoring
- Accident recognition
- Monitoring the elderly
- Data collection
- **Example: Intel**
    - 130 sensor nodes
    - Monitor the activity of elderly patients
    - Data is acquired with a wearable sensing nodes

Domotic

<u>Local and remote management of house</u>

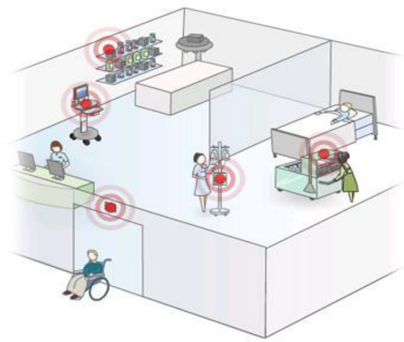
- Automation of appliances
- Brightness
  - Wireless switches
  - Window and blinds control
- Temperature
  - Wireless thermostats



<u>House Control</u>

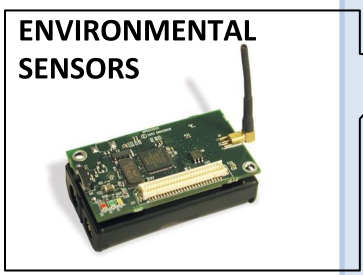
- Door control
- Security alarm

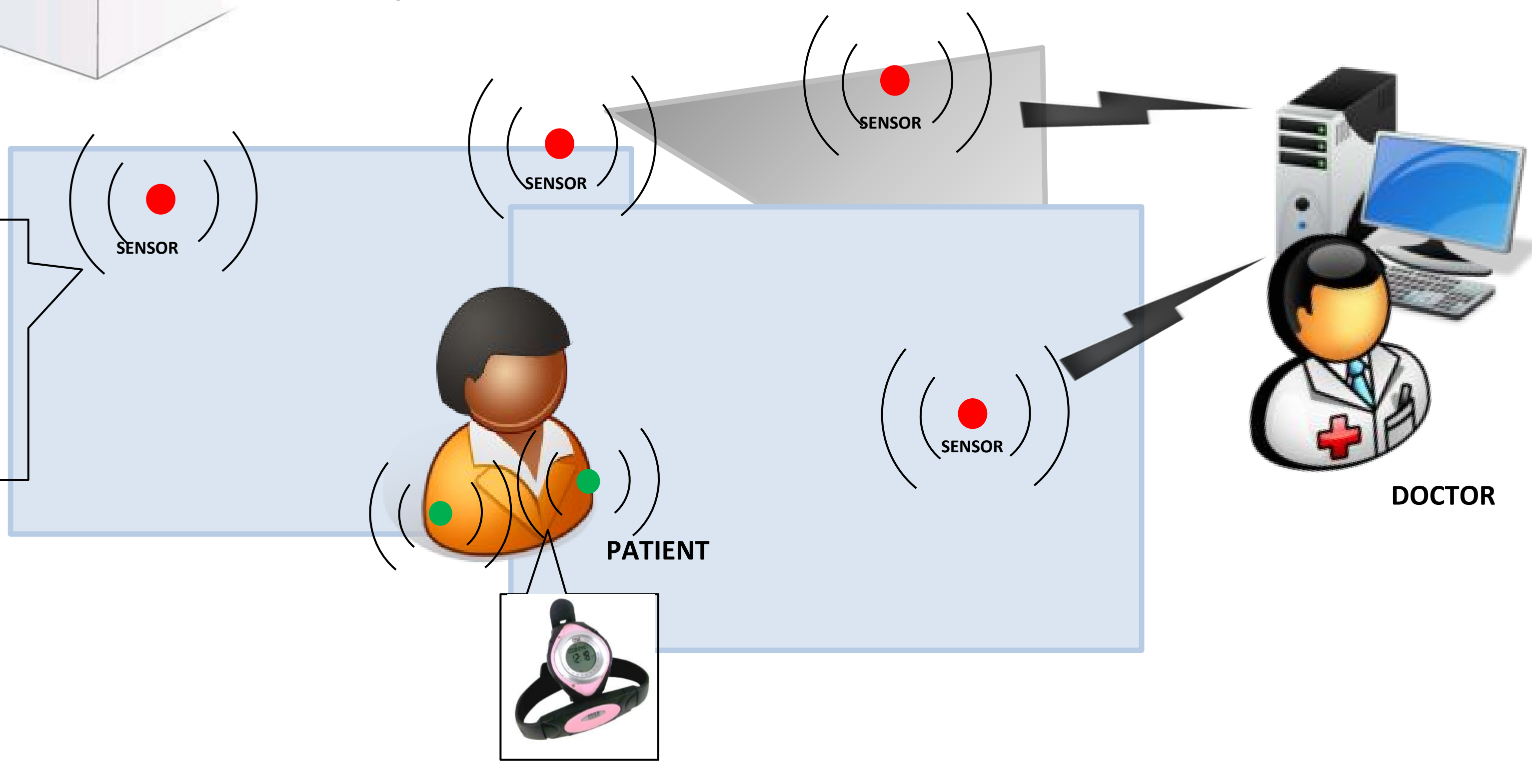
CHIRON project



***Aim**: person-centric health management*

- Correlation between the health status of patients and the environmental conditions



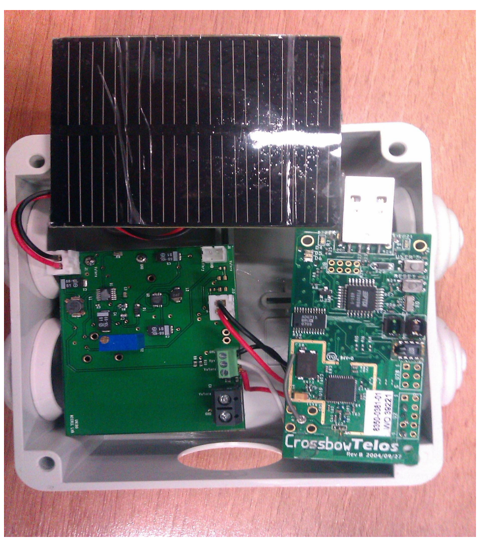


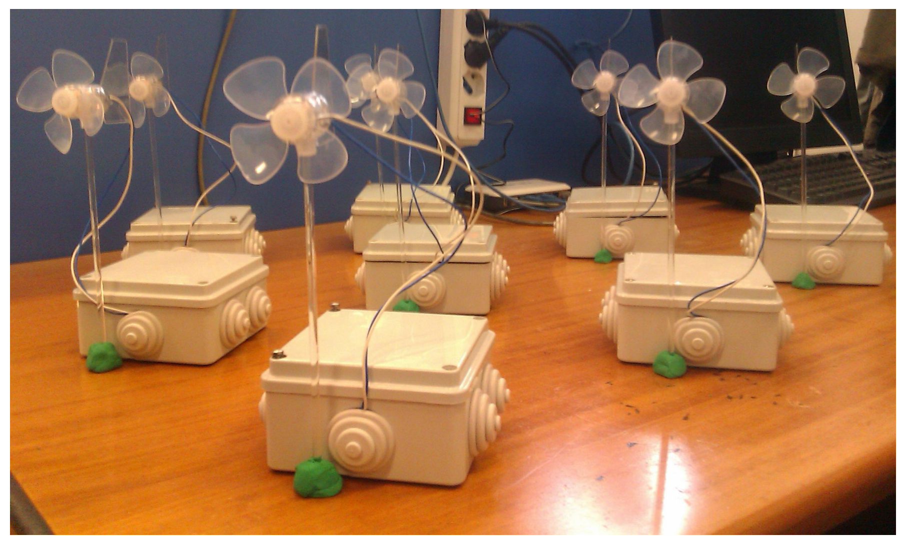
- **Light**
- **Temperature**
- **Humidity**
- **Pollution**

GENESI project

Aim: *Long-lasting sensing systems for Structural Health Monitoring*

- Green wireless sensor networks equipped with energy harvesting and triggering capabilities
- Use of energy from renewable resources to extend the network lifetime

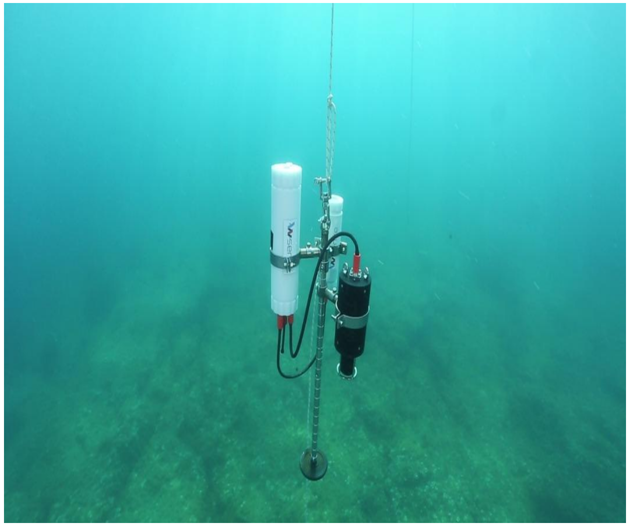
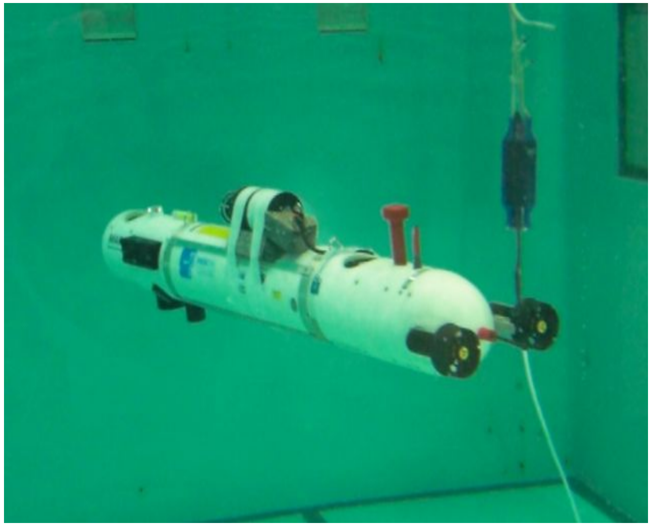




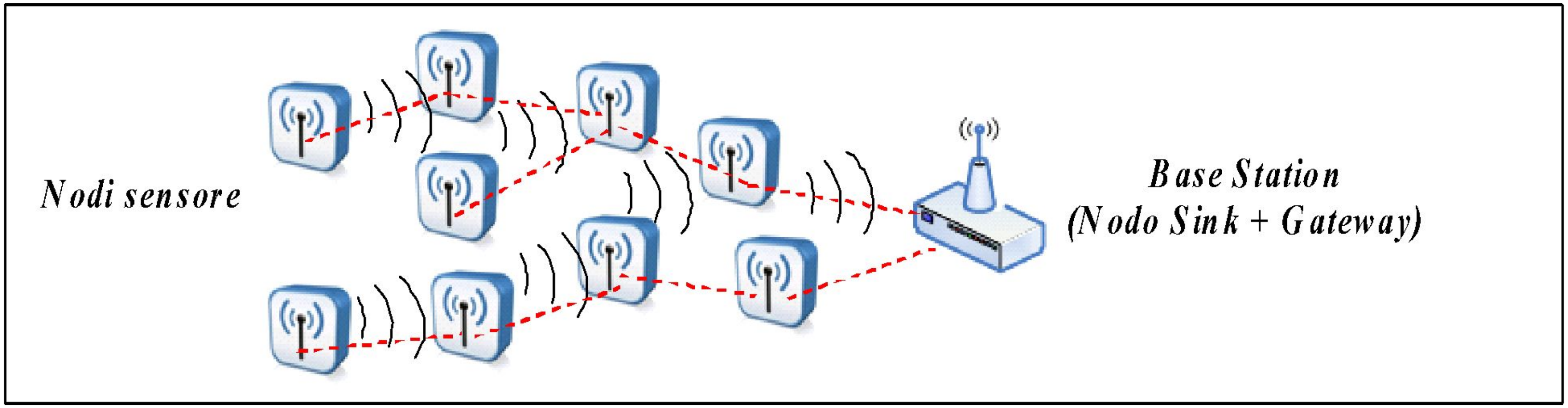
SUNRISE project

Aim: *Internet of Underwater Things*

- Sensing, monitoring, and actuating of the underwater communication networks
- Monitoring of oil, gas, CO2
- Prevention of natural disasters
- Chemical composition seabeds

WSN Architecture



- Autonomous devices (*sensor nodes*) geographically distributed
- Equipped with sensors
- Collaborate to monitor the surrounding environment
- Key elements
  - Sensor Node (*node, mote*) and *Base Station*
  - Short-range *wireless* communication *(multi-hop)*

WSN Characteristics

- Autoconfiguration
    - Manual configuration just not an option
- Scalability
    - Support large number of nodes
- Programmability
    - Re-programming of nodes in the field might be necessary, improves flexibility

WSN Characteristics

- *Low cost*
  - Number of nodes in WSNs is high; to make deployments possible, the nodes should be extremely low cost
- *Energy efficient*
  - Both form communucation and computation, sensing, actuating

Design Aspects

- **System model**
  - Physical nodes vs. Functional components
  - Local computation vs. Communication
- **Hardware architecture of nodes**
  - Microprocessor/Microcontroller
    - IBM 8051, Atmel ATmega128L, XScale PXA271, TI MSP430,…
  - Chipset for communication and the related antenna
    - ChipCon CC1100 and CC2420
  - Bus of local communication
    - SPI, I2C

Design Aspects

- **Communcation Protocols**
  - Diverse and heterogeneous protocols
  - Lower levels rely on standard protocols
    - IEEE 802.15.4, 6lowpan, Bluetooth, etc..
- **Routing algorithms**
  - Specific SPIN (Sensor Protocols for Information via Negotiation)
  - Directed Diffusion
  - Rumor Routing
  - Q-RC (Q-learning Routing and Compression)
  - etc..

Energy Saving Solutions

- **Nodes**
  - Design components and architecture hw/sw
  - Mechanisms for (auto) power management
- **Network**
  - *Energy-aware* protocols
- **System**
  - *Energy-aware* applications

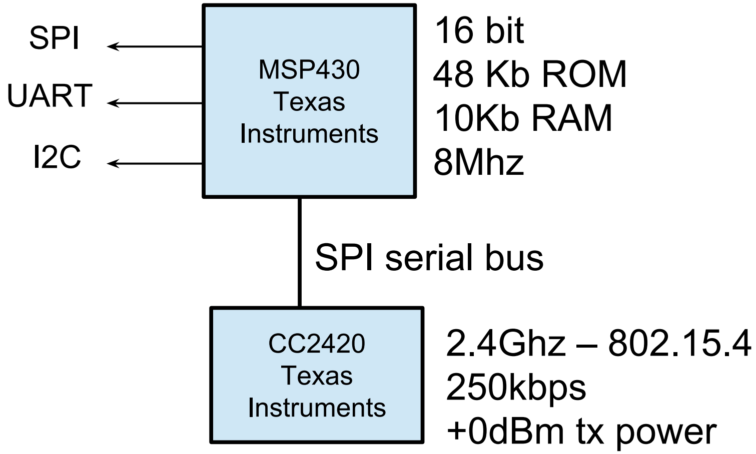
Security Solutions

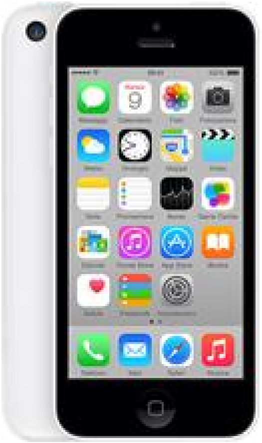
- **Nodes**
  - Encryption algorithms
  - Network
  - Cryptographic systems
  - Intrusion Detection & Monitoring Systems

- **System**
  - Cryptographic systems

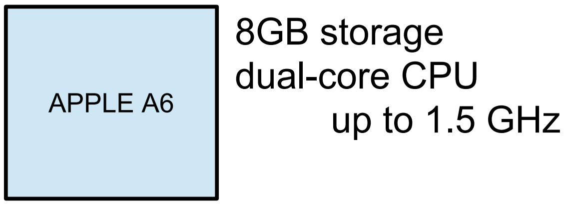
Hardware Characteristics

TelosB

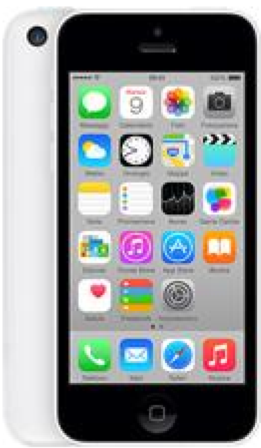




iPhone 5c



Energy Consumption



iPhone 5C
- Power: Non-removable Li-Po
- Expensive!
- Battery lasts < 1 day

TelosB
- Power: 2xAA alkaline batteries
- Cheap!

TinyOS

- ***What is TinyOS?***
  - A simple operating system for sensor networks and ambedded systems
  - Programming language is a C extention with extra features
  - Open source →
    - Source code easily reusable
    - Large developers community
  - Support for a great variety of hardware modules

- ***Why a new Operating System?***
  - Event-driven architecture; measure real-world phenomena
  - Resource constraints; **Hurry up and sleep!**
  - Hardware drivers, libraries, tools, compiler
  - Modular

TinyOS installation

Quickest option: install TinyOS via a virtual machine (VM)

1) Download and install VirtualBox
   - https://www.virtualbox.org

2) Download, untar and install TinyOS installation on the VM
   - https://mega.nz/#!ekQSHKaR!Z2_gKHnyNIIh5XhvdCxpail2LgcHM-FKdLWqQ1QSvZ0

3) Right click on the VirtualBox icon, and then Open
4) Last step is setting up the usb device on your VM
   - Devices->USB Devices-> XBOW Crossbow TelosRevB

TinyOS Installation

Alternative options

- **TinyOS** 2.1.2 installation
  - http://tinyos.stanford.edu/tinyos-wiki/index.php/Installing_TinyOS#Officially_Supported_Methods
  - http://tinyos.stanford.edu/tinyos-wiki/index.php/TinyOS_Tutorials
- **Linux:** .rpm and .deb packages for Fedora and Ubuntu
  - Recommend debian system installation on Ubuntu
- **Windows:** .rpm pkg, uses Cygwin to emulate Linux software layer
- **OS X:** Unofficially supported
  - http://tinyos.stanford.edu/tinyos-wiki/index.php/Installing_tinyos-2.x_on_Mac_OS_X_(Tiger_%26_Leopard)
  - https://olafland.wordpress.com/2012/06/25/tinyos-on-mac-os-x-10-7-lion/

TinyOS

- A library that includes nesC components and offers several functions like a common operating system:
  - **Scheduler**
  - **Driver**
    - Components for sensor data reading
    - Components for sending commands to actuators
    - Components for controlling radio communication
  - **Power Management**
    - Maintain available HW in the lowest possible power level
- **No kernel concepts, processes, memory management**

Native support for low-power operation

- Microcontroller Power Management
  - Microcontrollers should always be in the lowest power state possible
  - TinyOS handles state transitions automatically to achieve maximum power saving
- Radio Power Management
  - Duty-cycle radio to save energy and extend network lifetime
- Peripheral Energy Management
  - Energy-efficient scheduling of sensing operation and peripheral access

TinyOS

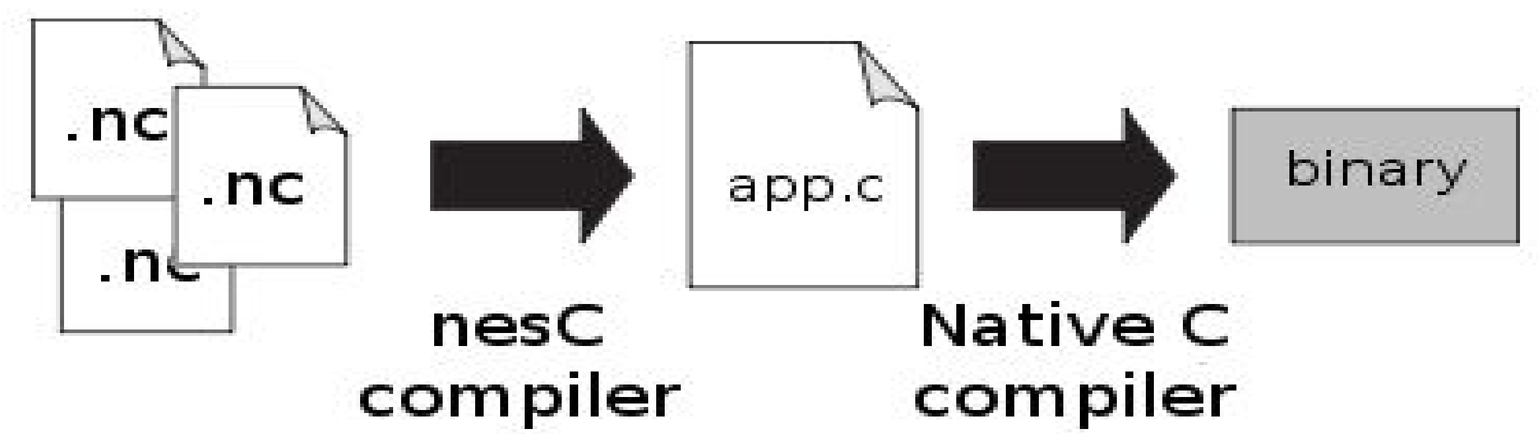
- For each application there is the *top-level* configuration that includes the *MainC* component
    - Provides services based on TinyOS (≈ 200 Bytes)

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

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

TinyOS-compiling

- TinyOS includes *Makefiles* to support the build process
- Create a *Makefile* in your application directory
    - COMPONENT = [MainComponentAppC] # the name of your AppC file



TinyOS-make system

- To compile an application without installing on a mote, run in the application directory:
    - make [platform] #ex. telosb

- To compile an application, and install it on a mote
    - make [platform] [re]install,[node ID] [programmingBoard,address]
    - node ID: 0-255, for radio transmissions
    - platform: defined in $TOOSROOT/tos/platforms
    - Programming board: for telosb use: bsl
    - Address: as reported by motelist
        - ex. /dev/ttyUSB0

TinyOS commands

- **motelist**
  - list of motes physically connected to your pc
- **make telosb**
  - compile your code for the telosb mote
- **make clean**
  - clean up all the compiled binary files
- **make telosb install,id bsl,address**
  - compile your code for telosb, install it on a mote, give it a network id
  - example: make telosb install,0 bsl,/dev/ttyUSB0
- **make telosb reinstall,id bsl,address**
  - use existing runnable, isntall in on telosb, give it a network id

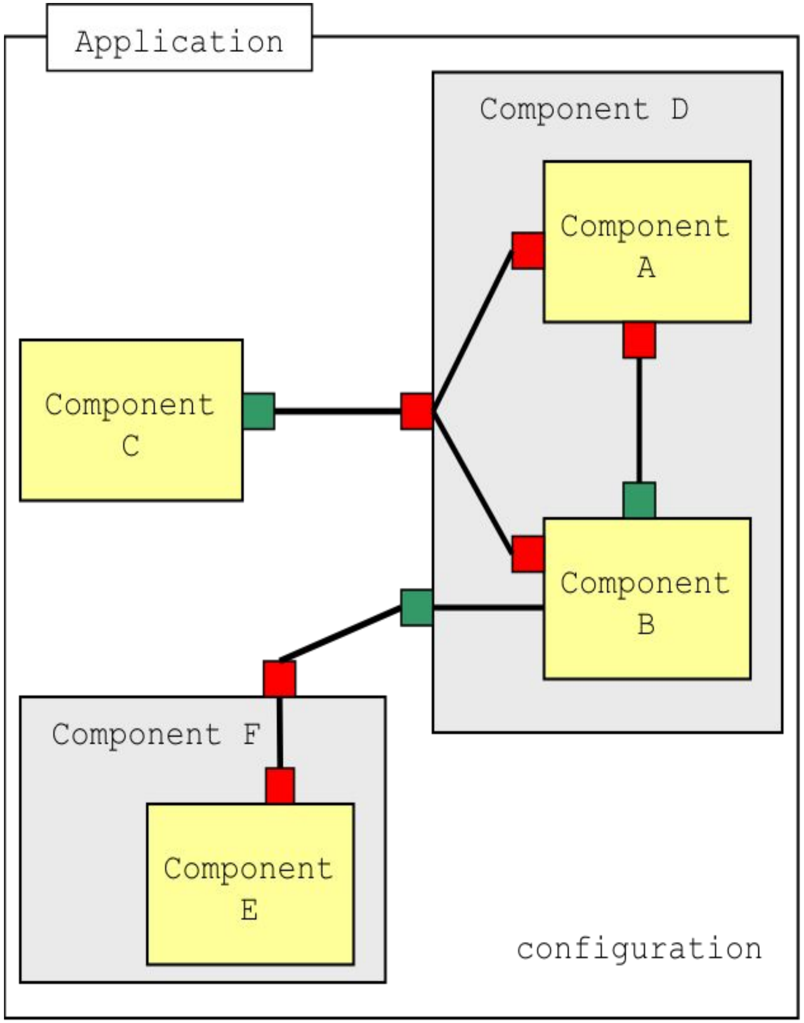
nesC Programming Language

How do we program wireless sensor devices

- ***nesC***
  - **Reference manual**
  http://nescc.sourceforge.net/papers/nesc-ref.pdf
- C extension language for networked embedded systems
- **Static language**:
  - no dynamic memory allocation
  - all resources known at compile-time

nesC Programming Language

- **Application**:one or more *components* are connected to each other (*wired*) to form an executable
- **Components:**
  - *Modules:* provide application code, implementing *interfaces*
  - *Configurations:* wire interfaces used by components to interfaces provided by others
- **Interfaces:** access to components
  - uses
  - provides



nesC Modules

- Provide/use one or more interfeces

```
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() {
     …
  }
}
```

nesC Configurations

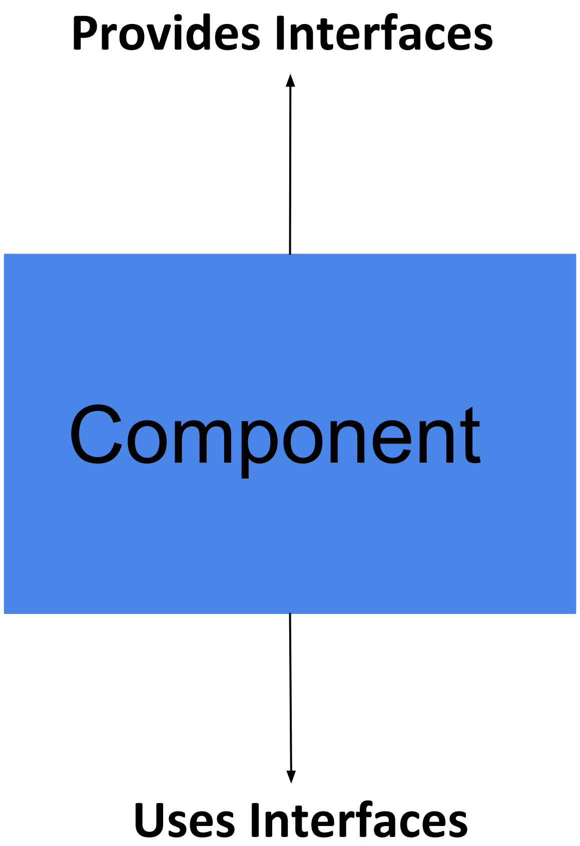
- Two components are linked together by *wiring* them
- Interfaces on user component are wired to the same interface on the provider component

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

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

nesC Interfaces

- ***Commands***: functions to be implemnted by the interface of the provider; how to use the interface
- ***Events***: functions to be implemnted by the interface of the user



nesC Concurrency

Two computational abstractions

Asynchronous events

- can run preemptively (async)
- interrupt handlers
- race conditions!

Tasks

- schedule a function to be called later
- run in a single execution context
- no preemption!
- FIFO

nesC Tasks

- Run sequential and to completion
- Do not preempt

```
task void computeTask() {
   uint32_t i;
   for (i = 0; i < 10001; i++) {}
}

event void Timer0.fired() {
   post computeTask();
   call Leds.led0Toggle();
}
```

nesC Events

- Run to completion; may preempt tasks and event
- Origin: hardware interrupts/split-phase completion

```
event void Boot.booted(){
   call Timer0.startPeriodic(250);
}
event void Timer0.fired() {
   post computeTask();
   call Leds.led0Toggle();
}
```

nesC Split-phase

- Enable TinyOS components to easily start several operations at once and have them executed in parallel.

Blocking	Split-Phase

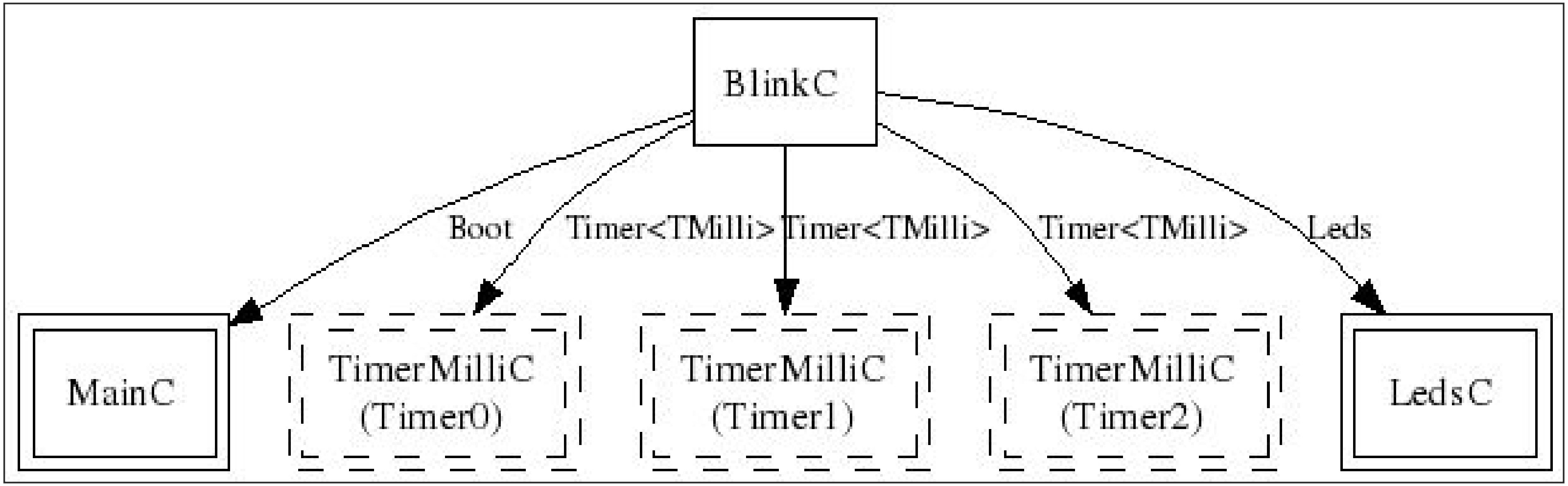
`state = WAITING;`
`op1();`
 	
`sleep(500);`
 	
`op2();`
`state = RUNNING;`
	`state = WAITING;`
	`op1();`
	
	`call Timer.startOneShot(500);`
	
	`event void Timer.fired() {`
	`    op2();`
	`    state = RUNNING;`
	`}`

Example: Blink Application

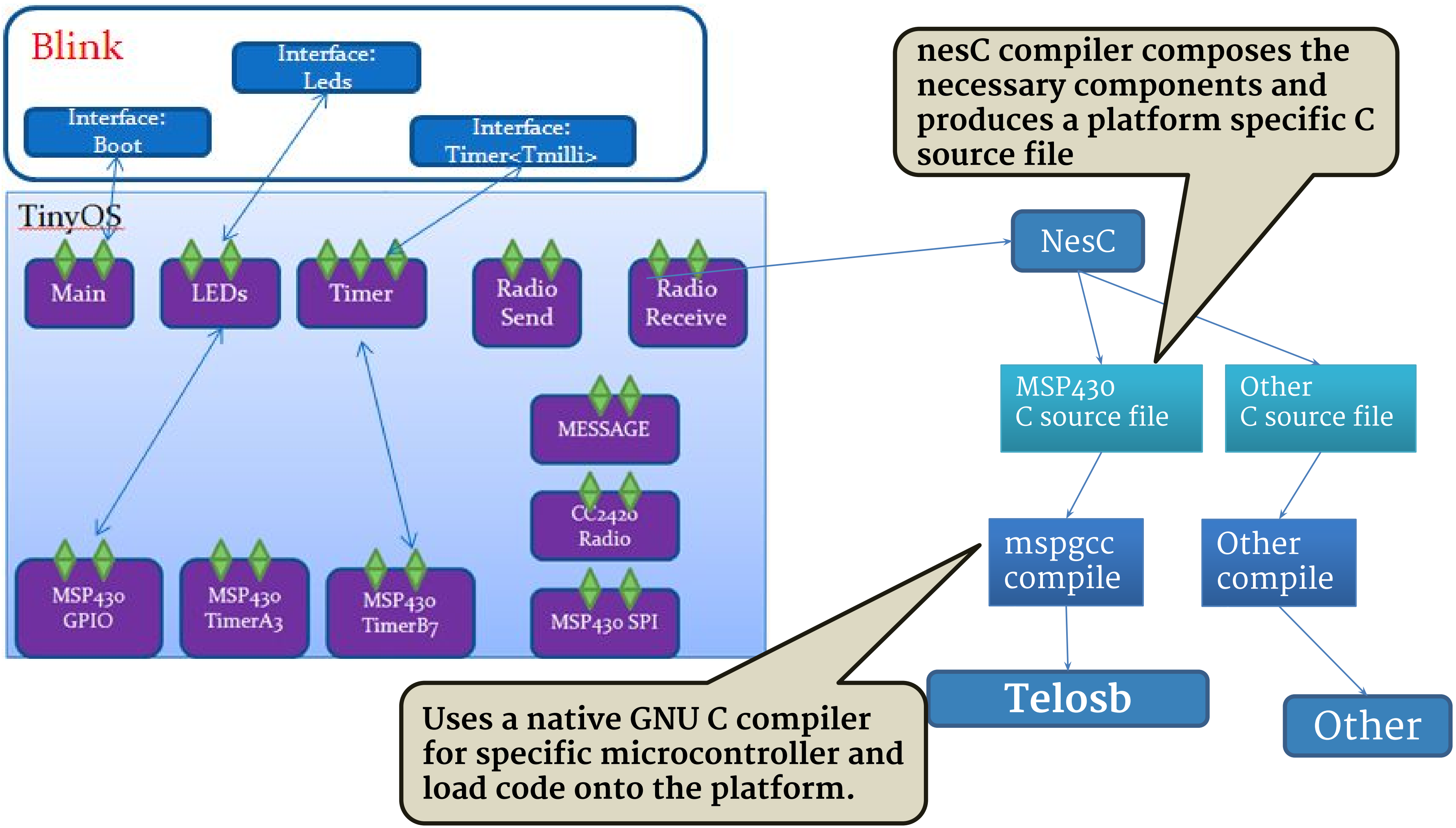
- /apps/Blink in the TinyOS tree
- Causes three LEDs to turn on and off
  - The LEDs turn on and off at the frequencies 1Hz, 2Hz, and 4Hz
- Application components
  - *BlinkAppC (Configuration)*
  - *BlinkC (Module)*
- System Components
  - *MainC, LedsC, TimerMilliC*

BlinkApp Components

- ***BlinkAppC***: Component graph
  - Single box: module, double box: configuration
  - Dashed lines: generic component



Blink Compilation



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
{
```

BlinkC.nc

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

Example: Blink Timer

- ***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();
  }
}
```

Exercise

Modify the Blink application

- Use only one timer firing once per second
- When the timer fires, increment a counter
- Display the value of the counter using the LEDs

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();
     }
}
```