Contents

Politecnico di Milano

- Introduction to WSNs
  - Generalities about WSNs and their components
  - Typical applications for WSNs
  - Routing in WSNs
  - MAC protocols for WSNs
- **Programming WSNs with TinyOS**
  - **Introduction to TinyOS and nesC**
  - **Fundamentals of TinyOS**
    - **Blink: The first TinyOS application**
  - **TinyOS advanced**
    - **Parameterized interfaces**
    - **Generic components**
    - **Concurrency**
    - **Using the radio**
    - **BlinkToRadio: An advanced example**

TinyOS: An OS for tiny devices

Politecnico di Milano

- TinyOS provides the minimal system support needed to write WSNs applications
  - More a library (of components) than an OS
  - Allows for fine grained resource management
- Offers an event-driven computational model
  - Well suited to the concurrency usually found in WSN applications
- Builds on the component abstraction
  - Allows for strong code reuse

Politecnico di Milano

The nesC language

- TinyOS system, libraries and applications are written in nesC
  - A component-based C dialect especially designed to code embedded systems
  - Provides constructs for defining, building, and linking components
  - Supports the TinyOS concurrency model
    - *Split-phase* operations and *tasks*... no threads
- nesC is the way the TinyOS programming model is expressed

Politecnico di Milano

TinyOS components

- A component is an encapsulated unit of functionality
- A component *provides* and *uses* interfaces
- *Interfaces* express what functionality are offered/used
  - Used interfaces represent functionality the component relies upon
  - Provided interfaces represent offered functionality that other components will rely upon
  - E.g., a component **`AODV`** provides the **`Routing`** interface and uses the **`MAC`** interface
- Components describe how the (provided) functionality is actually implemented (taking advantage of the used functionality)

Component syntax

Politecnico di Milano

Declaration of provided and used interfaces

Implementation of the component's functionality

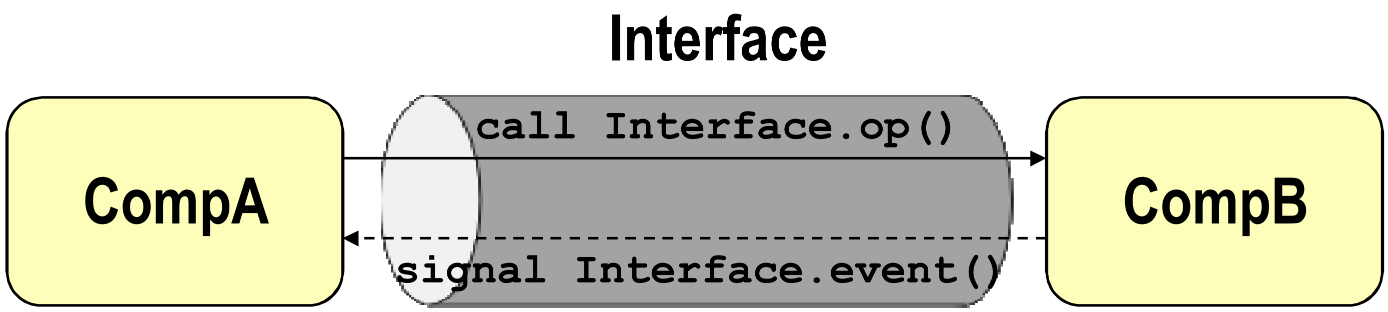
```
module FooM {
  provides {
    interface Foo;
  }
  uses {
    interface Poo;
    interface Boo;
  }
}

implementation {
  // Application code
}
```

TinyOS interfaces

Politecnico di Milano

- An interface is a collection of *commands* and *events*
- TinyOS interfaces are bidirectional
  - Commands are implemented by the component providing the interface
  - Events are implemented by the component using the interface
- For a component to call the commands in an interface, it must implement the events of the same interface



Interface syntax

Politecnico di Milano

A command implemented by the interface provider

An event implemented by the interface user

```
interface Foo {
  command int op1(params…);

  event void event1Fired();

  command result_t op2(params…);
  event void op2Done();
}
```

A split-phase operation, i.e., a non-blocking operation whose completion is signaled asynchronously with a corresponding event

`result_t` is a built-in type of nesC simply comprising `FAIL` or `SUCCES`

Interfaces with arguments

- Interfaces can take types as arguments

```
interface Read<val_t> {
  command error_t read(error_t err, val_t t);
  event void readDone();
}
```

- Modules providing/using such interfaces specify the type they need

```
module MagnetometerC {
  provides interface Read<uint16_t>;
}
```

- When wiring providers and users of typed interfaces their types must match
  - E.g., you cannot wire a `Read<uint8_t>` to a `Read<uint16_t>`

Modules vs. Configurations

Politecnico di Milano

- TinyOS provides two types of components: Modules and Configurations
- Modules are basic components, whose implementation is provided in C
  - Standard C constructs can be used to implement a component, including *calling* the commands exported by the interfaces it uses and *signalling* their events
  - A module must implement every command of interfaces it provides and every event of interfaces it uses
- Configurations are complex components that *wire* together other components
  - Connect interfaces used by some components to interfaces provided by others
  - Allow for hiding the implementation of a single service implemented with multiple interconnected components
    - e.g. a communication service that needs to be wired to timers, random number generators and low-level hardware facilities can be exported by means of a single configuration
- Configurations connect the *declaration* of different components, while modules *define* components by defining functions and allocating state

Modules

Politecnico di Milano

```
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t lastVal = 0;
  command error_t StdControl.start() {
    return call Timer.startPeriodic(1024);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
    }
  }
}
```

Configurations

```
configuration LedsC {
  provides interface Init();
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;
  Init = LedsP;
  Leds = LedsP;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

Basic nesC types

- Numeric types
  - Signed and unsigned integers
    - `int8_t   int16_t   int32_t`
    - `uint8_t  uint16_t   uint32_t`
  - Reals
    - `float   double`
- Other types
  - Characters
    - `char`
  - Booleans
    - `bool  (TRUE – FALSE)`
  - Errors
    - `error_t`
- Platform dependencies
  - Platform independent types: `nx_###`
    - E.g. `nx_uint16_t`
  - Platform independent structs can be defined with the `nx_struct` keyword and should include platform independent fields, only

Politecnico di Milano

Coding conventions

- Component and interface names follow the same convention of Java classes
- Command and event names follow the same convention of Java methods
- Internal variables and parameters follow the C convention
- Types are small caps ending with "_t"
- Private vs. public components
  - If a component is a usable abstraction by itself, its name should end with C
  - If it is intended to be an internal and private part of a larger abstraction, its name should end with P
  - Never wire to P components from outside your package (directory)

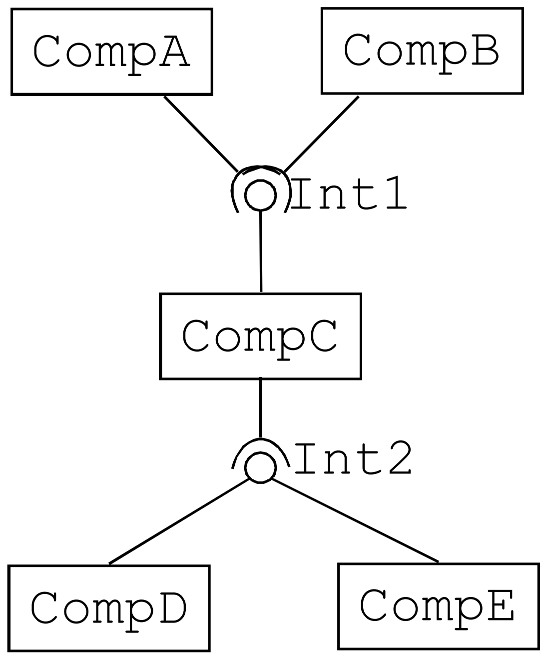
Politecnico di Milano

Components vs. classes

- Components (especially modules) are similar to classes in an OO language
  - They encapsulate a state (within their variables) and provide some functions
- But there is a big difference: you cannot instantiate them
  - *Components (like hardware components) are singletons*
- If two configurations in your code wire the same component they are wiring the same (and unique) instance of such component
  - As it happens with hardware components
- Consequence: the interface of a component can be wired many times to different components
  - *Calling a command and raising an event may result in invoking several components*

Multiple wirings and combine functions

- What if `CompC` raises an event part of `Int1` or it calls a command part of `Int2`?
  - Several components are invoked
  - The order is non deterministic
- What if the event or the command have a result value?
  - Results are combined using the *combine function* associated to the type of the result




```
typedef uint8_t error_t @combine("ecombine");

error_t ecombine(error_t e1, error_t e2) {
   return (e1 == e2)? e1: FAIL;
}
```

Politecnico di Milano

Application setup and startup

- The `Init` interface:

```
interface Init {
   command error_t init();
}
```

  should be provided by components that need to be initialized before the application starts
- The `Boot` interface:

```
interface Boot {
   event void booted();
}
```

  should be used by the top-level component that represent the nesC application, to be notified when everything has been initialized (e.g., to start timers)
- Component `MainC` provides `Boot` and uses `Init` (as `SoftwareInit`)
  – It should be wired to every component needing to be initialized
- The `StdControl` interface:

```
interface StdControl {
   command error_t start();
   command error_t stop();
}
```

  should be provided by components that need to be started/stopped at run-time

Application setup and startup

```
module FooP {
  provides {
    interface Init;
    interface SplitControl;
    ...
  }
  uses { ... }
}
implementation { ... }
```

```
configuration FooC {
  provides {
    interface SplitControl;
    ...
  }
}
implementation {
  components MainC, FooP, ...;
  MainC.SoftwareInit -> FooP;
  SplitControl = FooP.SplitControl;
  ...
}
```

```
module TestC {
  uses {
    interface Boot;
    interface SplitControl as FooCont;
    ...
  }
}
implementation {
  event void Boot.booted() {
    call FooCont.start();
  }
  event void FooCont.startDone(error_t
   e) {
    ...
  }
}
```

```
configuration TestAppC {}
implementation {
  components MainC, TestC, ...;
  TestC.Boot -> MainC.Boot;
  ...
}
```

Blink: The main module

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

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

  event void Timer1.fired() { call Leds.led1Toggle(); }

  event void Timer2.fired() { call Leds.led2Toggle(); }
}
```

Politecnico di Milano

Blink: The top-level configuration

```
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;
}
```

Blink: Building the application

Politecnico di Milano

The Makefile

```
COMPONENT=BlinkAppC
include $(MAKERULES)
```

- Compiling for telosb
  `make telosb`
- Listing the connected motes
  `motelist`
- Installing on a node with network id 10
  ```
  make telosb reinstall,10
    bsl,/dev/ttyUSB0
  ```
  ```
  make telosb reinstall,10
    bsl,1
  ```
- Compiling for TOSSIM
  `make micaz sim`