POLITECNICO
MILANO 1863

Middleware Technologies for Distributed Systems

Programming WSNs with TinyOS

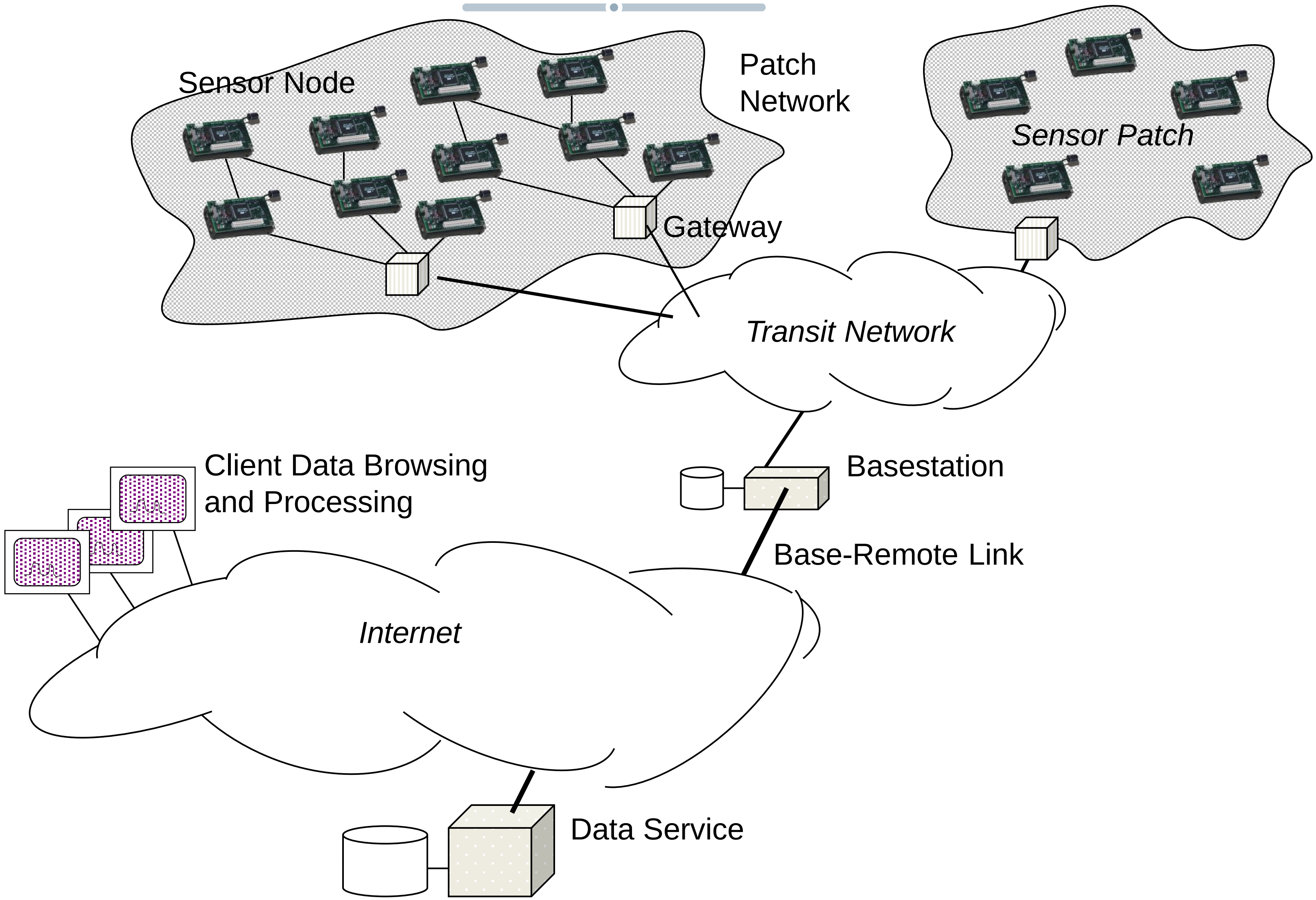
Gianpaolo Cugola

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano, Italy

gianpaolo.cugola@polimi.it
http://home.dei.polimi.it/cugola

Contents

The architecture of a typical WSN



WSN: Distinctive Features

- Variable number of nodes, sometime very large
  - #sensor nodes >> #users
  - Nodes need to be close to each other
  - Nodes often cannot have a global ID such as an IP address
- Non-interactive, data-centric and event-centric applications,
  - Asymmetric flow of information, from sensor nodes to sink
  - Most often, content-based rather than identity-based communication
  - The identity of the single node does not matter to the user: unlikely to perform queries for a single sensor node
- The network is application-specific
  - In contrast with traditional networks
  - Often, support for aggregation and other features is provided as part of the network functionality
  - More "cross layering" than ISO stack

WSN: Distinctive Features

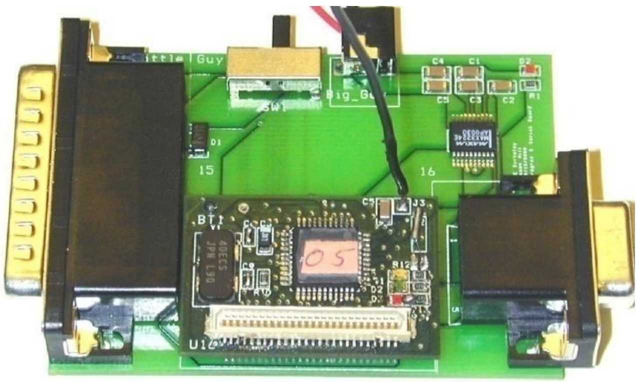
- Deployed in uncontrolled, often inaccessible and/or hostile environments
  - Sensors must self-organize (e.g., topology)
  - Must guarantee "exception-free", unattended operation
  - Also often impossible to replace energy source
- Energy usage must be carefully managed, to increase the lifetime *of the whole network*
  - Sensors typically "sleep" most of the time, and wakeup shortly to perform their sensing tasks as well as communication
    - MICA mote always on: ~6 days; 2% duty cycle (1.2s/min): 6 months
  - In-network data aggregation is often supported as a means to reduce traffic overhead

WSN: Distinctive Features

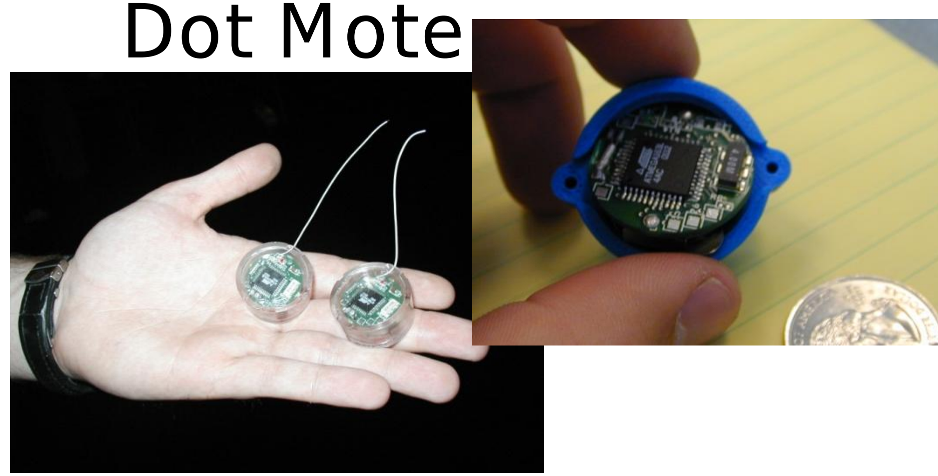
- Usually, “closed”, fixed environment: sensors rarely move
  - Not true for mobile sensor networks (e.g., for wildlife monitoring) and other applications (e.g., health, body area networks)
  - In any case, *fixed sensors but dynamic topology*
    - Due to sleep patterns/failures
- In many applications:
  - Need for real-time delivery (e.g., for monitoring) and time synchronization (to properly correlate data)
  - Need for k-coverage, i.e., ensure that every point in a given area is in range of at least k sensors

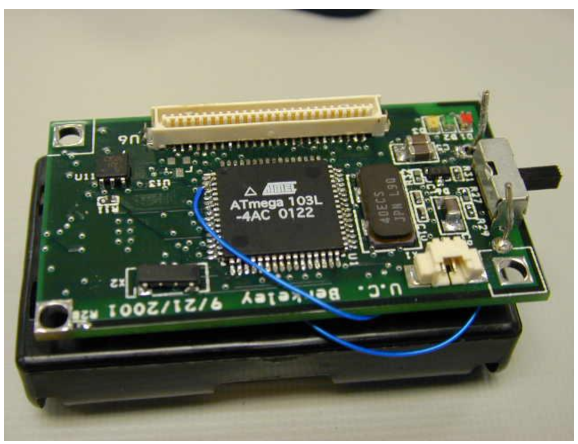
Examples of sensor nodes

Rene Mote

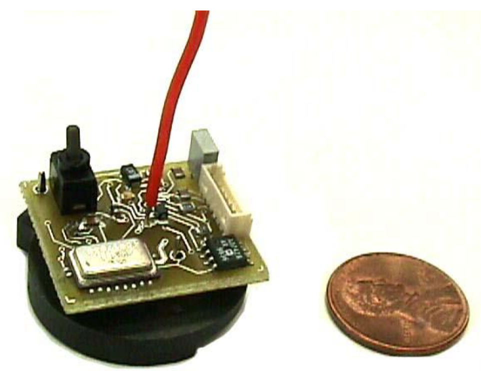


Dot Mote





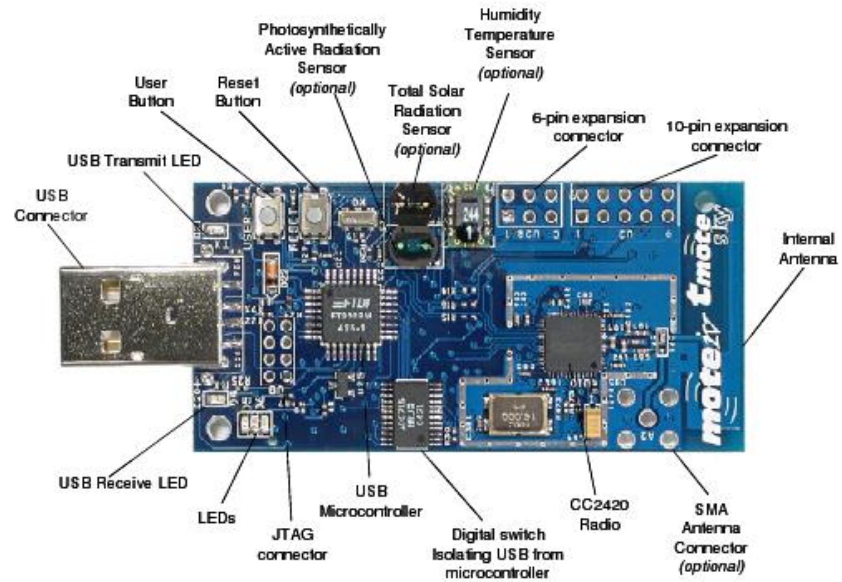
MICA Mote

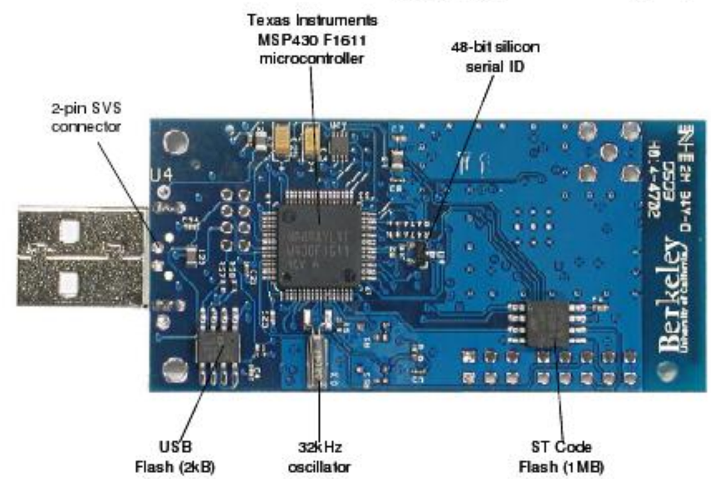


weC Mote

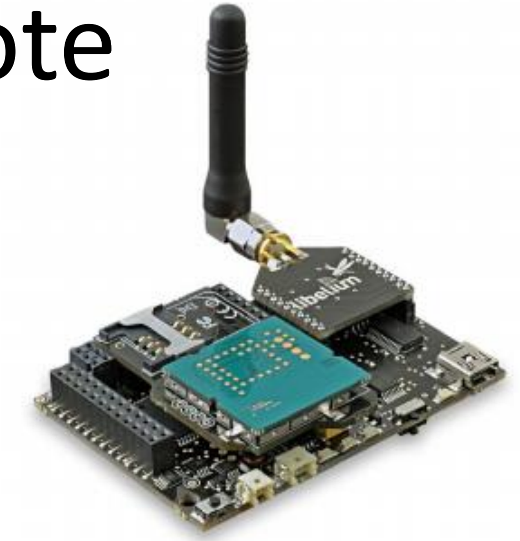
TelosB/TMoteSky

- Open Source platform designed by UCBerkeley
  - Two producers (crossbow and moteiv), two names (TelosB and TMoteSky)
- Powered by two AA batteries
- Texas Instruments MSP430 µC
  - 16bit, RISC, 8 MHz
  - 10KB RAM, 48Kbyte program flash
  - 1Mbyte of external flash memory
- TI CC2420 multichannel radio
  - 802.15.4 compliant
  - ISM band (2.4 GHz)
  - Bandwidth 250 kbits/s
  - Integrated antenna: Range of up to 150m
- Sensors:
  - Default: light (solar and IR), humidity, temperature
  - Other: 6 ADC and 2 DAC external ports at 12 bit

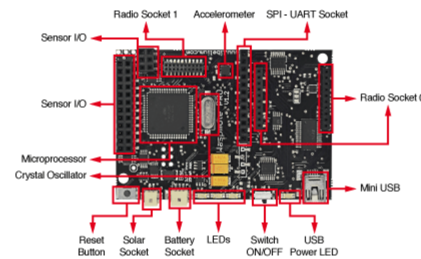


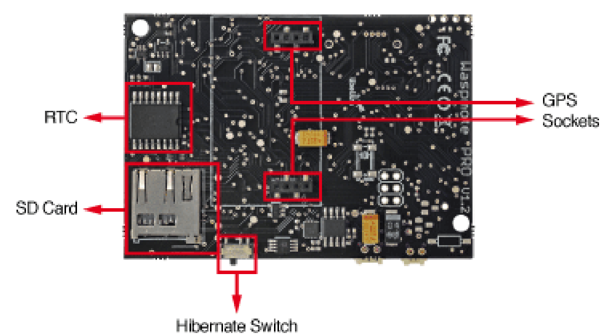


Libelium Waspmote



- Commercial platform designed by libelium
- ATmega1281 µC
  - 16bit, RISC, 14 MHz
  - 8KB RAM, 4Kbyte program flash
  - 128Kbyte of external flash memory+SD
- Radio (several through daughter boards)
  - 802.15.4 compliant (short and long range)
  - Long range module based on 868-915 MHz transmitter
  - Bluetooth Low Energy (BLE) 4.0
  - WiFi
  - 3G/GPRS/GSM
  - RFID/NFC
- Built-in sensors:
  - Temperature (+/-): -40ºC , +85ºC. Accuracy: 0.25ºC
  - Accelerometer: ±2g/±4g/±8g
- Additional sensors available through (tens of) daughter boards





Applications

- Military
  - Monitoring friendly forces, equipment, and ammunition
    - Enhanced logistics systems
  - Battlefield surveillance
    - Intrusion detection (mine fields)
    - Reconnaissance of opposing forces and terrain
  - Targeting and target tracking systems
    - Detection of firing gun (small arms) location
  - Nuclear, Biological and Chemical (NBC) attack detection and reconnaissance
- Environmental
  - Monitoring environmental conditions that affect crops and livestock
    - Precision agriculture
  - Biocomplexity mapping of the environment
    - Tracking the movements of birds, small animals, and insects
  - Earth monitoring and planetary exploration
    - Meteorological or geophysical research
  - Chemical/biological detection
    - Pollution study
  - Flood detection, and forest fire detection

Applications

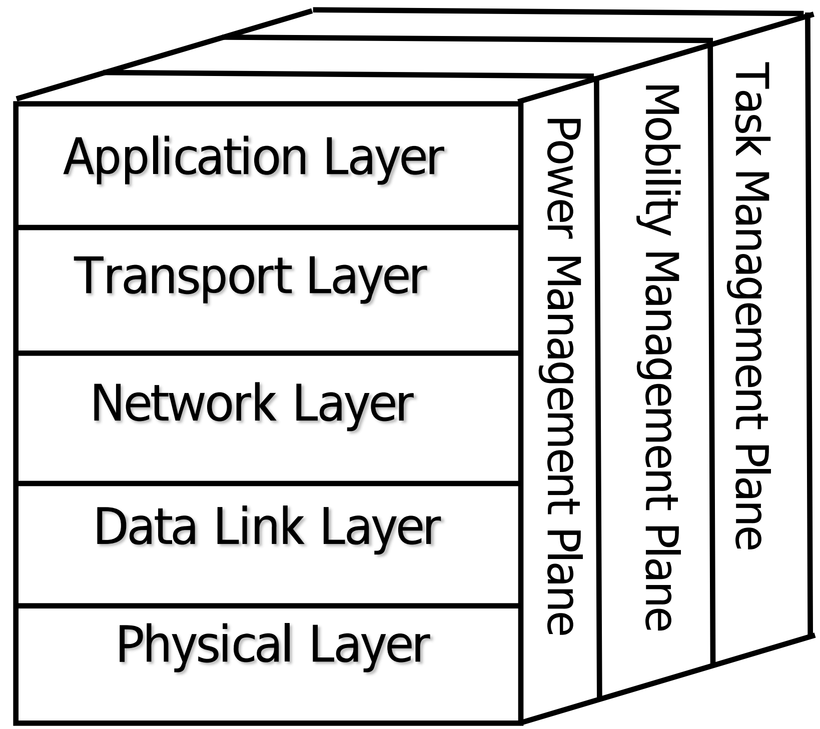
- Healthcare
  - Tracking and monitoring doctors and patients inside a hospital
  - Drug administration in hospitals
  - Diagnostics
  - Telemonitoring of human physiological data
  - Providing interfaces for the disabled
- Others
  - Factory Automation
    - Monitoring product quality
    - Machine diagnosis
    - Managing inventory control
  - Smart home/office
    - Environmental control in office buildings
  - Interactive toys
  - Monitor disaster areas
  - Interactive museums

Underwater Sensor Networks

- Typically based on ultrasound
- Applications:
  - Pollution monitoring and other environmental monitoring (chemical, biological)
  - Disaster prevention
  - Assisted navigation
  - Distributed tactical surveillance (mine reconnaissance)

Communication Architecture

- Used by sink and all sensor nodes
- A lot of cross layering
  - Combines power and routing awareness...
  - ... to communicate power efficiently through wireless medium
  - Integrates data with networking protocols
- Promotes cooperative efforts



Routing in WSNs

- Why not use conventional routing algorithms?
  - Unique node addresses cannot be used in many sensor networks
    - Sheer number of nodes, energy constraints, data centric approach
  - Sensor nodes are very limited in power, computational capacities, and memory
  - They are very prone to failures
  - WSN topology changes frequently
  - Nodes are densely deployed
  - Routing in WSNs need tight integration with sensing tasks

Taxonomy of Routing Protocols

- Data-centric protocols
  - Flooding, Gossiping, SPIN, SAR (Sequential Assignment Routing) , Directed Diffusion, Rumor Routing, Constrained Anisotropic Diffused Routing, COUGAR, ACQUIRE
- Hierarchical protocols
  - LEACH, TEEN (Threshold Sensitive Energy Efficient Sensor Network Protocol), APTEEN, PEGASIS
- Location-based protocols
  - MECN, SMECN (Small Minimum Energy Com Netw), GAF (Geographic Adaptive Fidelity), GEAR, Distributed Topology/Geographic Routing Algorithm

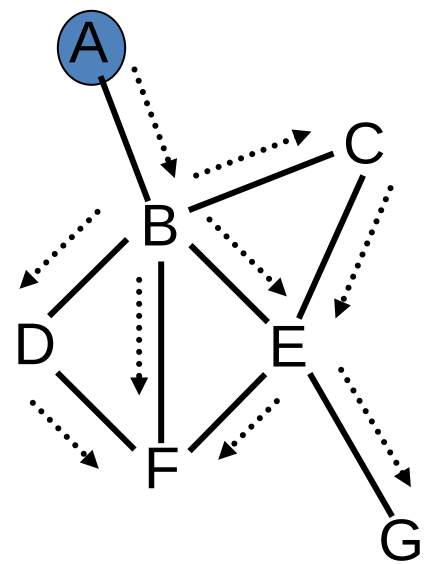
K. Akkaya and M. Younis, "A Survey on Routing Protocols for Wireless Sensor Networks," AdHoc Networks (Elsevier) Journal, 2005

Data-Centric Routing

- Since sensor nodes are deployed randomly in large number, it is hard to assign specific IDs to each of the sensor nodes
    - Without a unique identifier, gathering data may become a challenge
- To overcome this challenge, some routing protocols gather/route data based on the description of the data, i.e., data-centric
- The data-centric routing requires attribute based naming where the users are more interested in querying an attribute of the phenomenon, rather than querying an individual node.
    – Example: "the areas where the temperature is over 70F“ is a more common query than "the temperature read by a certain node”

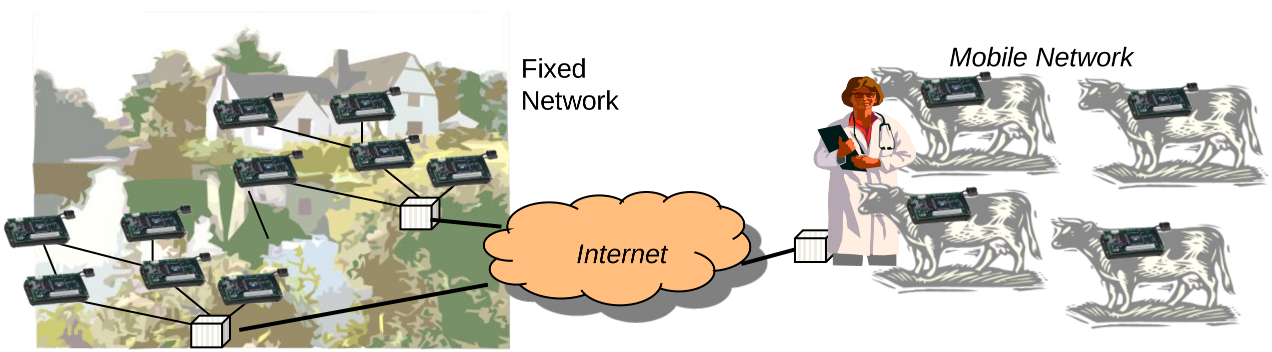
Data Dissemination Approaches

- Flooding
  - Send data to all neighbors
- Gossiping
  - Send data to one randomly selected neighbor
- Both are simple, but…
  - Implosions
    - Same data reaching the same node several times (through different routes)
  - Overlaps
    - Same data read by different nodes
  - Resource blindness (e.g., remaining battery)
    - Energy inefficiency



Mobile & multi-sink WSNs

- WSNs are mature to be applied to new scenarios
  - Monitoring people (e.g., elderly care, ...), animals (e.g., in farms, ...), mobile things in general (e.g., in logistics, ...)
- *Mobility* is the distinctive characteristic of all these scenarios
  - Mobility of sensors and/or sinks
- Moreover, in advanced scenarios we cannot ignore the presence of *multiple sinks*
  - One or more gateways toward fixed networks
  - PDAs in the hand of operators roaming around
  - Actuators acting as sinks



CCBR: Context and content-based routing for multi-sink mobile WSNs

- Interactions in WSNs are *data-centric*...
  - WSNs are designed to gather data and deliver it
- ... but also *context-aware*
  - A farmer could be interested in having the temperature reading (each hour) of young cattle only
  - In a logistics application different data must be collected for different kind of products

We need a context & content based routing protocol (CCBR)

CCBR: The API

`setProperties(…)`

- Advertizes the node's context, e.g., kind=cow & age=12

`listenFor(ComponentFilter, MessageFilter, …)`

- Chooses the relevant nodes and messages to receive

`send(…)`

- Sends out messages

The CCBR protocol

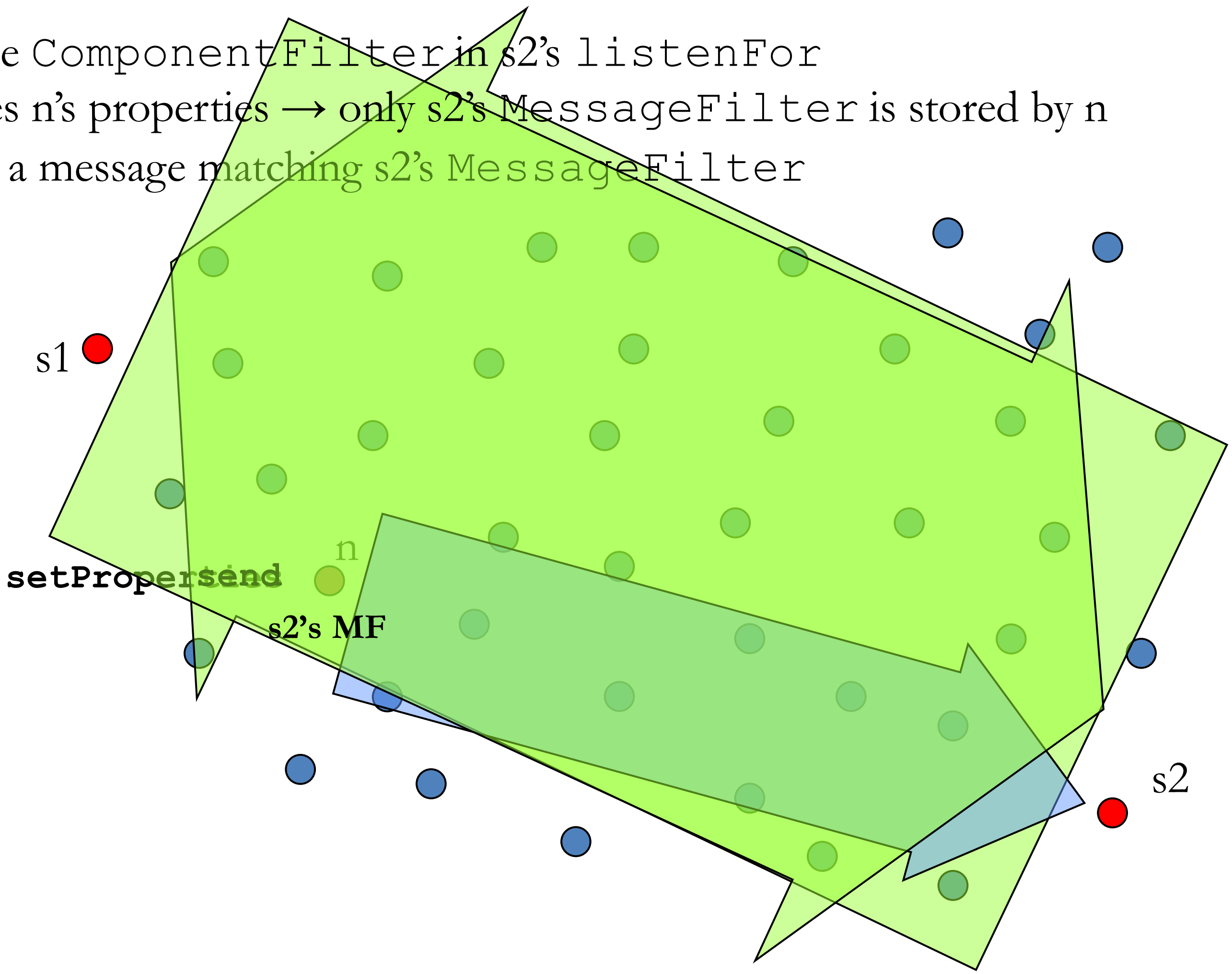
- Nodes keep track of their distance from the "sinks"
  - i.e., those nodes that issued a `listenFor`
- Nodes' properties are stored locally
  - No communication
- MessageFilters flood the network...
  - Piggybacked on beacons used to update distances from the "sinks"
- ...and they are stored only by the nodes having the "right" component properties
  - Reduce memory usage
- Messages are routed only toward the interested sinks
- Content-based matching is done at the sender node
  - Reduce CPU usage

The CCBR protocol (cont.d)

s1 and s2 issue a `listenFor`

Only the `ComponentFilter` in s2's `listenFor` matches n's properties → only s2's `MessageFilter` is stored by n

n sends a message matching s2's `MessageFilter`



The CCBR protocol (cont.d)

- Use *link-layer broadcast* whenever possible
  - To be robust against mobility
  - To minimize traffic when multiple sinks have to be reached
  - Taking advantage of TrawMAC ability of optimizing power usage for broadcast communication
- Use a *receiver-based* approach
  - The receiving node decides if accepting the packet and re-forwarding it, not the sender
- Use an *opportunistic* approach
  - Each node hearing a packet *locally* decides if re-forwarding it...
  - ...using its estimate distance from the sinks it decides if forwarding the packet and how long to *delay* it before forwarding...
  - ...if the same packet is heard again the delayed packet is *thrown away*
- Overhearing is used as an *implicit ack*
  - An initial number of *credits* is assigned to packets to limit re-forwarding due to missed acks

Hierarchical Protocols

- Hierarchical-architecture protocols are proposed to address the scalability and energy consumption challenges of sensor networks
- Sensor nodes form clusters where the cluster-heads aggregate and fuse data to conserve energy
  - The cluster-heads may form another layer of clusters among themselves before reaching the sink

Location-Based Protocols

- If the locations of the sensor nodes are known, the routing protocols can use this information
  - To reduce the latency and...
  - ... energy consumption of the sensor network
- Although GPS is not envisioned for all types of sensor networks, it can still be used if stationary nodes with large amount of energy are allowed
- Other protocols exists to calculate the location of nodes based on RSSI and triangulation

Challenges in MAC for WSNs

- WSN architecture
  - High density of nodes
  - Increased collision probability
  - Signaling overhead should be minimized to prevent further collisions
  - Sophisticated and efficient collision avoidance protocols required
- Limited Energy Resources
  - Need very low power MAC protocols
  - Transmitting and receiving consumes almost same energy
    - Minimize signaling overhead
    - Avoid idle listening
  - Frequent power up/down eats up energy
    - Prevent frequent radio state changes (active←→sleep)
- Limited Processing and Memory Capabilities
  - Complex algorithms cannot be implemented
  - Conventional layered architecture may not be appropriate
    - Cross-layer optimization required
  - Centralized or local management is limited
    - Self-configurable, distributed protocols required

Challenges in MAC for WSNs

- Limited bandwidth and packet size
  - Limited header space
    - Unique node ID is not practical, local IDs should be used for inter-node communication
  - MAC protocol overhead should be minimized
- Cheap Encoder/Decoders
  - Cheap node requirement prevents sophisticated encoders/decoders
    - Simple FEC codes required for error control
  - Cheap node requirement prevents expensive crystals to be implemented
    - Inaccurate clock crystals → synchronization problems
    - TDMA-based schemes are hard to implement
- Event-based Networking
  - Observed data depends on physical phenomenon
  - Spatial and temporal correlation in the physical phenomenon should be exploited

Existing MAC protocols cannot be used for sensor networks!!!!

Contention-based MAC protocols

- Every node tries to access the channel based on carrier sense mechanism
- Contention-based protocols provide robustness and scalability to the network
- The collision probability increases with increasing node density
- They can support variable, but highly correlated and dominantly periodic traffic

IEEE 802.15.4

- The MAC of the ZigBee stack
  - A standard de-facto, nowadays
- In non-beacon mode does not provide any mechanism to reduce power consumption
  - It adopts a pure unslotted, contention based, CSMA/CA approach
  - Each node have to continuously listen the channel
- Some variants have been proposed (e.g., in TinyOS) called "Low Power Listening" MAC
  - They allow nodes to adopt their own sleep/wakeup schedule
  - Using long preambles to synchronize senders with receivers
  - Provide best performance when communication is very rare

Reservation-based MAC protocols

- These are collision-free MAC protocols!
  - Each node transmits data to a central agent during its reserved slot
- The duty cycle of the nodes is decreased resulting in further energy efficiency
- Generally, these protocols follow common principles where the network is divided into clusters and each node communicates according to a specific super-frame structure
- The super-frame structure consists of two main parts
  - The *reservation period* is used by the nodes to reserve their slots for communication through a central agent, i.e., cluster-head
  - The *data period* consists of multiple slots, used by each sensor for transmitting information
- The contention schemes for reservation protocols, the slot allocation principles, the frame size, and clustering approaches differ in each protocol

Contention-based vs. reservation-based protocols

- Contention-based protocols provide better scalability and lower delay, when compared to reservation-based protocols
- On the other hand, the energy consumption is significantly higher than the TDMA-based approaches due to collisions and collision avoidance schemes
- Contention-based protocols are more adaptive to the changes in the traffic volume and hence applicable to applications with bursty traffic such as event-based applications
- TDMA-based protocols require an infrastructure consisting of cluster heads which coordinate the time slots assigned to each node
  - The synchronization and clustering requirements of reservation-based protocols make contention-based more favorable in scenarios where such requirements can not be fulfilled
- Finally, time synchronization is an important part of the TDMA-based protocols and synchronization algorithms are required

Cross Layer Solutions

- Incorporating physical layer and network layer information into the MAC layer design improves the performance in WSN
- Route-aware protocols provide lower delay bounds while physical layer coordination improve the energy efficiency of the overall system
- Moreover, since sensor nodes are characterized by their limited energy capabilities and memory capacities, cross-layer solutions provide efficient solutions in terms of both performance and cost
- However, care must be taken while designing cross-layer solutions since the interdependence of each parameter should be analyzed in detail

Operating System Support

- Peculiarities:
  - Small memory footprint
  - Intrinsically reactive operation
  - Highly modular
  - Distinction between layers becomes blurred
    - Tight integration of the different layers
    - The network layer becomes the most relevant
    - Often poor memory management and no file system
    - Typically provided as a library, which must be linked with the application
- Examples:
  - TinyOS
  - Contiki
  - Mantis
  - SOS

Contents

- 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

- 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

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

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

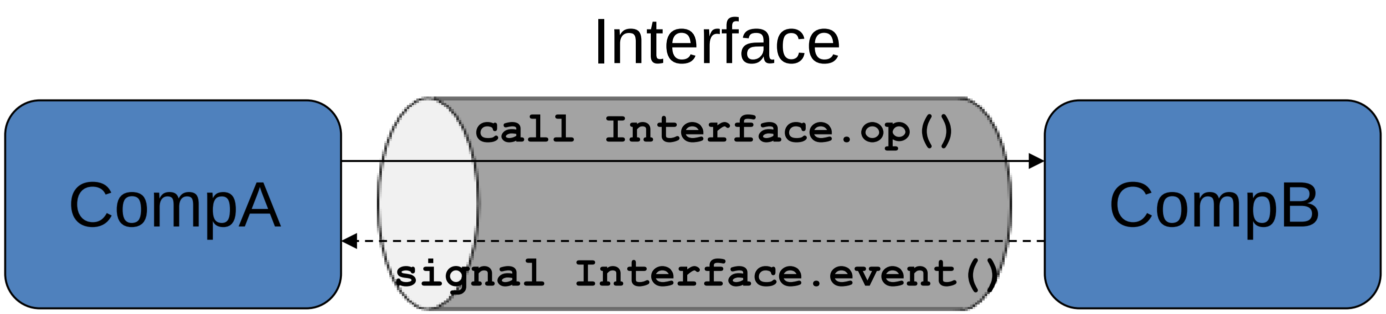
Declaration of provided and used interfaces

Implementation of the component's functionality

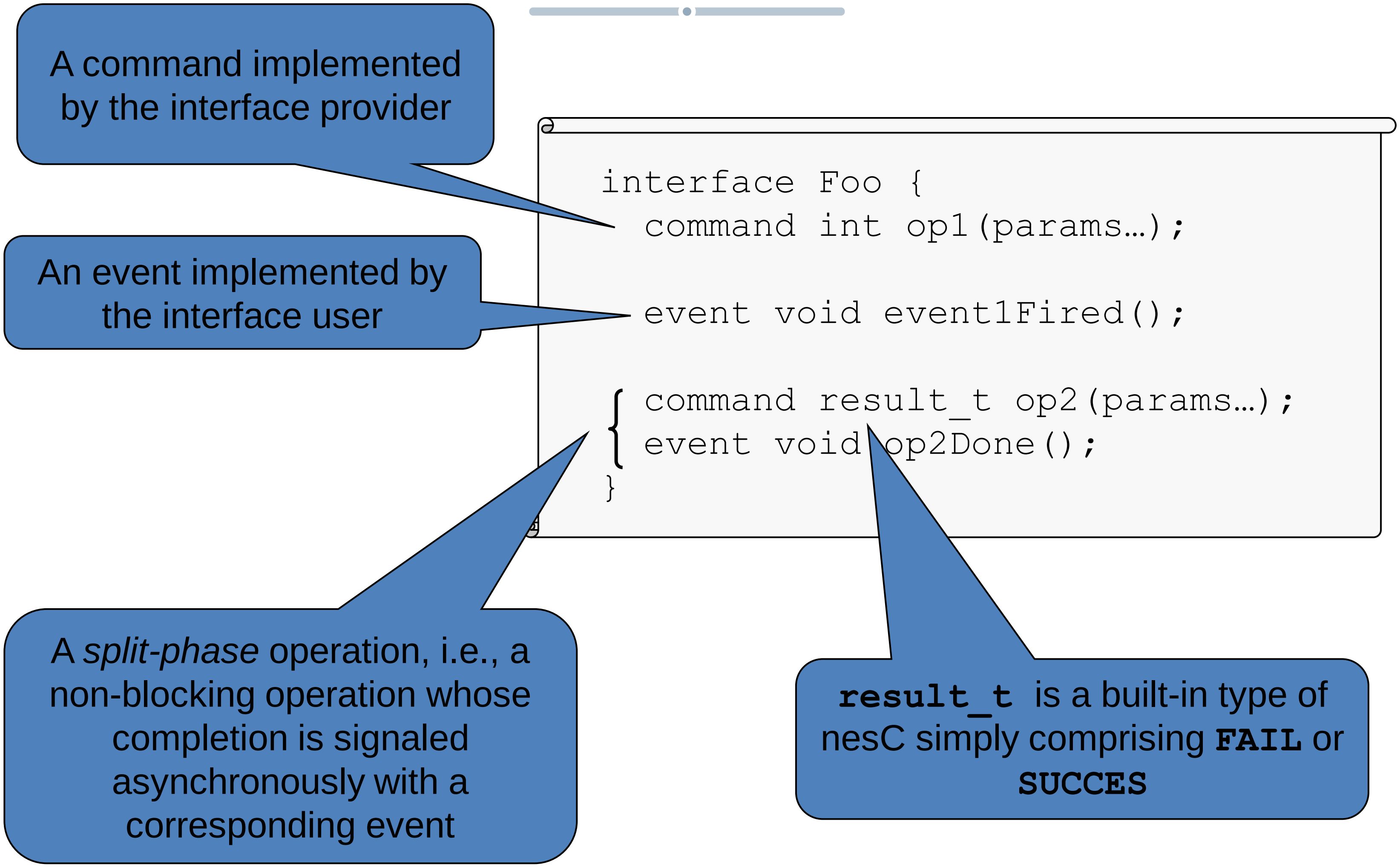
```
module FooM {
  provides {
    interface Foo;
  }
  uses {
    interface Poo;
    interface Boo;
  }
}

implementation {
  // Application code
}
```

TinyOS interfaces

- 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



Interfaces with arguments

- Interfaces can take types as arguments

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

- 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

- 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

```
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

Coding conventions

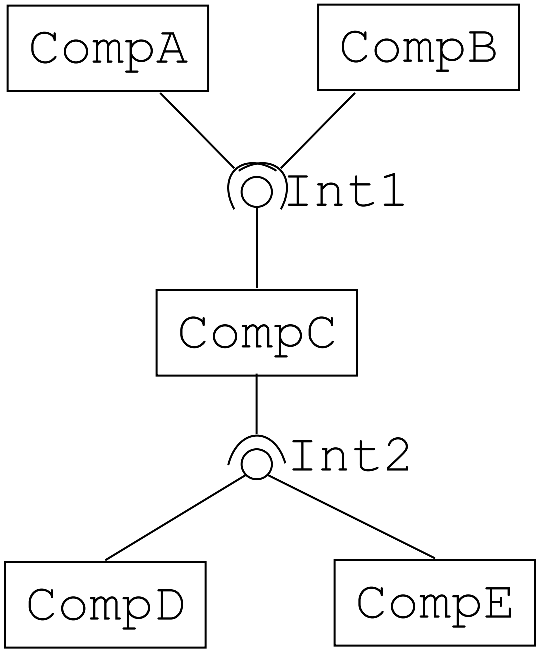
- Component and interface names follow the same convention of Java classes
- Command and event names follow the same convention of Java methods
- Internal variables and parameters follow the C convention
- Types are small caps ending with "_t"
- Private vs. public components
  - If a component is a usable 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)

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;
}
```

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(); }
}
```

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

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
  ```

A ready to use VirtualBox VM for tinyos is available at:

https://drive.google.com/open?id=1epXJQDRl08akSSc5BFBjHu0EBs0xbtXu

TOSSIM

- TOSSIM (TinyOS SIMulator) allows TinyOS applications to be simulated on a standard machine
  - It works by replacing components with simulation implementations
- TOSSIM is a library: you must write a program that configures a simulation and runs it
  - TOSSIM supports two programming interfaces: Python and C++
- Using TOSSIM and Python (or C++) you can:
    - Start, interrupt, and restart the simulation
    - Use debug statements in your code
    - Define the physical position of your motes and the characteristics of the radio channel
- For more information see the tutorial at: http://tinyos.stanford.edu/tinyos-wiki/index.php/TOSSIM

Parameterized interfaces

- Sometimes, a component wants to provide many instances of an interface
  - For example, the basic timer implementation component `HilTimerMilliC` needs to provide many timers
- Two possibilities:
  - Name the different interfaces explicitly

```
configuration HilTimerMilliC {
  provides interface Timer<TMilli> as Timer0;
  provides interface Timer<TMilli> as Timer1;
  ...
  provides interface Timer<TMilli> as Timer100;
}
```

  - Add a parameter to the interface functions to identify the timer instance

```
interface MultiTimer<precision_tag> {
  command void startPeriodic(uint8_t timerId, uint32_t dt);
  command void stop(uint8_t timerId);
  event void fired(uint8_t timerId);
}
configuration HilTimerMilliC {
  provides interface MultiTimer<TMilli>
}
```

- Both solutions are poor
  - E.g.: What happens to events fired by `HilTimerMilliC` in the second case?

Parameterized interfaces

- A parameterized interface is essentially an array of interfaces
  - The array index is the parameter implicitly passed to the functions of the interface
- Example (taken from the TOSSIM implementation)

```
module HilTimerMilliC {
  provides interface Timer<TMilli> as TimerMilli[uint8_t num];
}
implementation {
  command void TimerMilli.startPeriodic[uint8_t num]( uint32_t dt ) {
    ...
  }
  command void TimerMilli.stop[uint8_t num]() {
    ...
  }
}
```

- What happens when the event fires?

```
...
signal TimerMilli.fired[num]();
...
```

the nesC compiler takes care of dispatching the event to the "right" component, i.e., to the one that wired to the interface number id

Parameterized interfaces: Wiring

- Wiring parameterized interfaces

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

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> HilTimerMilliC.TimerMilli[0];
  BlinkC.Timer1 -> HilTimerMilliC.TimerMilli[1];
  BlinkC.Timer2 -> HilTimerMilliC.TimerMilli[2];
  BlinkC.Leds -> LedsC;
}
```

- What if other components wire to the 0^{th} interface?
  - Use the `unique` function, which is resolved at compile time by the nesC compiler

```
BlinkC.Timer0 ->
  HilTimerMilliC.TimerMilli[unique("TimerMilli")];
```

Parameterized interfaces: Defaults

- When a component provides a parameterized interface it acts as it provides an array of interfaces…
- …what happens to the events associated to the elements of this array that are not actually used?
- As an example, consider an application that uses only two timers from the HilTimerMilliC component
  - NesC requires that all the possible fired events are wired (not only 0 and 1)
- Solution: the component may provide a “default” implementation for the events
  - Example

```
module HilTimerMilliC { ... }
implementation {
  ...
  default event void TimerMilli.fired[uint8_t id]() {}
}
```

Generic components

- Standard components are singletons. Generic components are not. They can be instantiated in a configuration using the operator new
  - Instantiation results in effectively copying the component code
- Generic components differ in syntax from singleton components by having an argument list
  - Example

```
generic module BitVectorC(uint16_t max_bits) { ... }

generic module VirtualizeTimerC(typedef precision_tag, int max_timers ) {
  provides interface Timer<precision_tag> as Timer[ uint8_t num ];
  uses interface Timer<precision_tag> as TimerFrom;
}

generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
implementation {
  components TimerMilliP;
  Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
}

configuration TimerMilliP {
  provides interface Timer<TMilli> as TimerMilli[uint8_t id];
}
implementation {
  components HilTimerMilliC, MainC;
  MainC.SoftwareInit -> HilTimerMilliC; TimerMilli = HilTimerMilliC;
}
```

The TinyOS concurrency model

- NesC functions (i.e., commands or events) may be synchronous or not
  - All the NesC functions we have seen so far are synchronous
  - Asynchronous commands or events must be explicitly marked with the `async` keyword
  - Example:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);
  ...
}
interface Leds {
  async command void led0On();
  async command void led0Off();
  ...
}
```

- Synchronous functions cannot interrupt each other...
  - They "run-to-completion"
- ...but they can be interrupted by asynchronous functions
- On the other hand, NesC enforces a *race-free invariant*:
  - State shared by asynchronous and synchronous code must be protected (in the asynchronous code) with the `atomic` statement

The TinyOS concurrency model

- Atomic blocks behave like synchronous functions: They run-to-completion
  - Example

```
async command bool increment() {
  atomic {
    a++;
    b = a + 1;
  }
}
```

- Rule: Commands called by an async function and events signaled by an async function must be async as well
- Conversely: Sync functions may freely call async commands or signal async events

TinyOS concurrency: Tasks

- TinyOS functions, being synchronous or not, may post a *task*
  - Tasks run atomically (like sync functions)...
  - ...under control of the TinyOS scheduler
- Since tasks are invoked asynchronously w.r.t. the caller (the component which post the task) they cannot have a return value
  - They act like deferred procedure calls
- Tasks execute within the naming scope of the defining component (they are private to such component)
  - They do not take any parameter: any parameter required can just be stored in the component
- Tasks are declared with:

```
task void taskname();
```

- They are defined with:

```
task void taskname(){
  // Task code
}
```

- They are queued for execution with

```
post taskname();
```

When using tasks

- In a synchronous function
  - Use a task to split a long running execution
    - To improve responsiveness
  - Use a task to emulate split-phase execution
    - Example

```
module filteredReader {
  provides interface Read<uint16_t>
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t> as RawRead;
}
implementation {
  uint16_t filteredVal = 0;
    // read periodically and calculate filtered value
    ...
  task void readDoneTask() {
    signal Read.readDone(SUCCESS, filteredVal);
  }
  command error_t Read.read() {
    post readDoneTask();
    return SUCCESS;
  }
}
```

- In an asynchronous function
  - Use a task to invoke synchronous code

The TinyOS concurrency model revisited

- NesC assumes an execution model that consists of run-to-completion tasks (that typically represent the ongoing computation), and interrupt handlers that are signaled asynchronously by hardware
- Because tasks are not preempted and run to completion, they are atomic with respect to each other, but are not atomic with respect to interrupt handlers
- Races are avoided either by accessing a shared state only in tasks, or only within atomic statements
  - The nesC compiler reports potential data races to the programmer at compile-time, thus enforcing the race-free invariant

Using the radio

- TinyOS provides direct access to the radio of the platform through several interfaces including:
  - `Packet`: Provides the basic accessors for the `message_t` abstract data type
    - Provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.
  - `Send`: Provides the basic address-free message sending interface
    - Provides commands for sending a message and canceling a pending message send
  - `Receive`: Provides the basic message reception interface
    - Provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area
  - `PacketAcknowledgements`: Provides a mechanism for requesting acknowledgements on a per-packet basis
- Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio...
- ... and to add addresses for destinations

The AM layer

- Packets used by the AM layer have an associated AM type
  - AM types are similar in function to the Ethernet frame type field, IP protocol field, and the UDP port in that all of them are used to multiplex access to a communication service
- AM packets also includes a destination field, which stores an "AM address" to address packets to particular motes
- Besides the standard interfaces to access the radio, the AM layer adds some more
  - `AMPacket`: Similar to `Packet`, provides the basic AM accessors for the `message_t` abstract data type
    - Provides commands for getting and setting a node's AM address, an AM packet's destination, and an AM packet's type
  - `AMSend`: Similar to `Send`, provides the basic Active Message sending interface
    - The key difference between `AMSend` and `Send` is that `AMSend` takes a destination AM address in its send command

The AM layer

- A number of components implement the basic communications and active message interfaces
  - `ActiveMessageC`: The main component
    - Provides all the interfaces below (`AMSend` as a parameterized interface), plus the `SplitControl` interface to start/stop the AM layer
  - `AMReceiverC`: A generic configuration
    - The parameter of the configuration is the AM type to use
    - Provides the following interfaces: `Receive`, `Packet`, and `AMPacket`
  - `AMSenderC`: A generic configuration (same as above)
    - Provides `AMSend`, `Packet`, `AMPacket`, and `PacketAcknowledgements` as `Acks`
  - `AMSnooperC`: A generic configuration (same as above)
    - Provides `Receive`, `Packet`, and `AMPacket`

BlinkToRadio: The header

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H

enum {
  AM_BLINKTORADIO = 6,
  TIMER_PERIOD = 250
};

typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;

#endif
```

BlinkToRadio: The main module

```
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
  uses interface Packet;
  uses interface AMPacket;
  uses interface AMSend;
  uses interface Receive;
  uses interface SplitControl as AMControl;
}
implementation {
  uint16_t counter;
  message_t pkt;
  bool busy = FALSE;

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

  event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call Timer0.startPeriodic(TIMER_PERIOD);
    }
    else {
      call AMControl.start();
    }
  }

  event void AMControl.stopDone(error_t err) {
  }

  event void Timer0.fired() {
    counter++;
    if (!busy) {
      BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)
        call Packet.getPayload(&pkt,
                    sizeof(BlinkToRadioMsg));
      if (btrpkt == NULL) return;
      btrpkt->nodeid = TOS_NODE_ID;
      btrpkt->counter = counter;
      if (call AMSend.send(AM_BROADCAST_ADDR,
          &pkt, sizeof(BlinkToRadioMsg)) ==
   SUCCESS) {
        busy = TRUE;
      }
    }
  }

  event void AMSend.sendDone(message_t* msg,
   error_t err) {
    if (&pkt == msg) busy = FALSE;
  }

  event message_t* Receive.receive(message_t* msg,
                     void* payload, uint8_t len){
    if (len == sizeof(BlinkToRadioMsg)) {
      BlinkToRadioMsg* btrpkt =
   (BlinkToRadioMsg*)payload;
      call Leds.set(btrpkt->counter & 0x03);
    }
    return msg;
  }
}
```

BlinkToRadio: The top-level configuration

```
#include "BlinkToRadio.h"

configuration BlinkToRadioAppC {
}
implementation {
  components MainC;
  components LedsC;
  components BlinkToRadioC as App;
  components new TimerMilliC() as Timer0;
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO);
  components new AMReceiverC(AM_BLINKTORADIO);

  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
  App.Packet -> AMSenderC;
  App.AMPacket -> AMSenderC;
  App.AMControl -> ActiveMessageC;
  App.AMSend -> AMSenderC;
  App.Receive -> AMReceiverC;
}
```

BlinkToRadio: Building and running in TOSSIM

- Compiling for TOSSIM:

```
make micaz sim
```

- Create the file describing the topology of the network

```
java net.tinyos.sim.LinkLayerModel topoConfig.txt
```

  - The `LinkLayerModel` class is part of TinyOS
- Run the application in TOSSIM using python

```
run.py
```

  - The `run.py` file is included into the code distributed as part of this lesson
- Suggestion: TOSSIM does not simulate leds. Use the `dbg` statement to have a trace of what is happening

Exercise

- Implement a flooding protocol for a multi-hop network
  - Node 0 periodically sends a packet in broadcast
    - Holding an identifier (a progressive number)
  - Receiving nodes (first time) forward it to their neighbors (in broadcast)
  - On receiving the same packet (same identifier) twice, a node drops the packet
- Periodically print the total number of packets received and the total number of packets forwarded
  - To calculate efficacy and efficiency of the protocol

References

- TinyOs Home Page
  - http://tinyos.stanford.edu/tinyos-wiki/
  - There you will find several tutorials and the NesC reference manual
- TinyOs is now hosted at GitHub:
  - https://github.com/tinyos/tinyos-main

- A ready-to-use VirtualBox VM for tinyos (made for the course) is available at:
  - https://drive.google.com/open?id=1epXJQDRl08akSSc5BFBjHu0EBs0xbtXu