Distributed Systems

Programming WSNs with TinyOS

Gianpaolo Cugola

Dipartimento di Elettronica e Informazione

Politecnico di Milano, Italy

cugola@elet.polimi.it

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

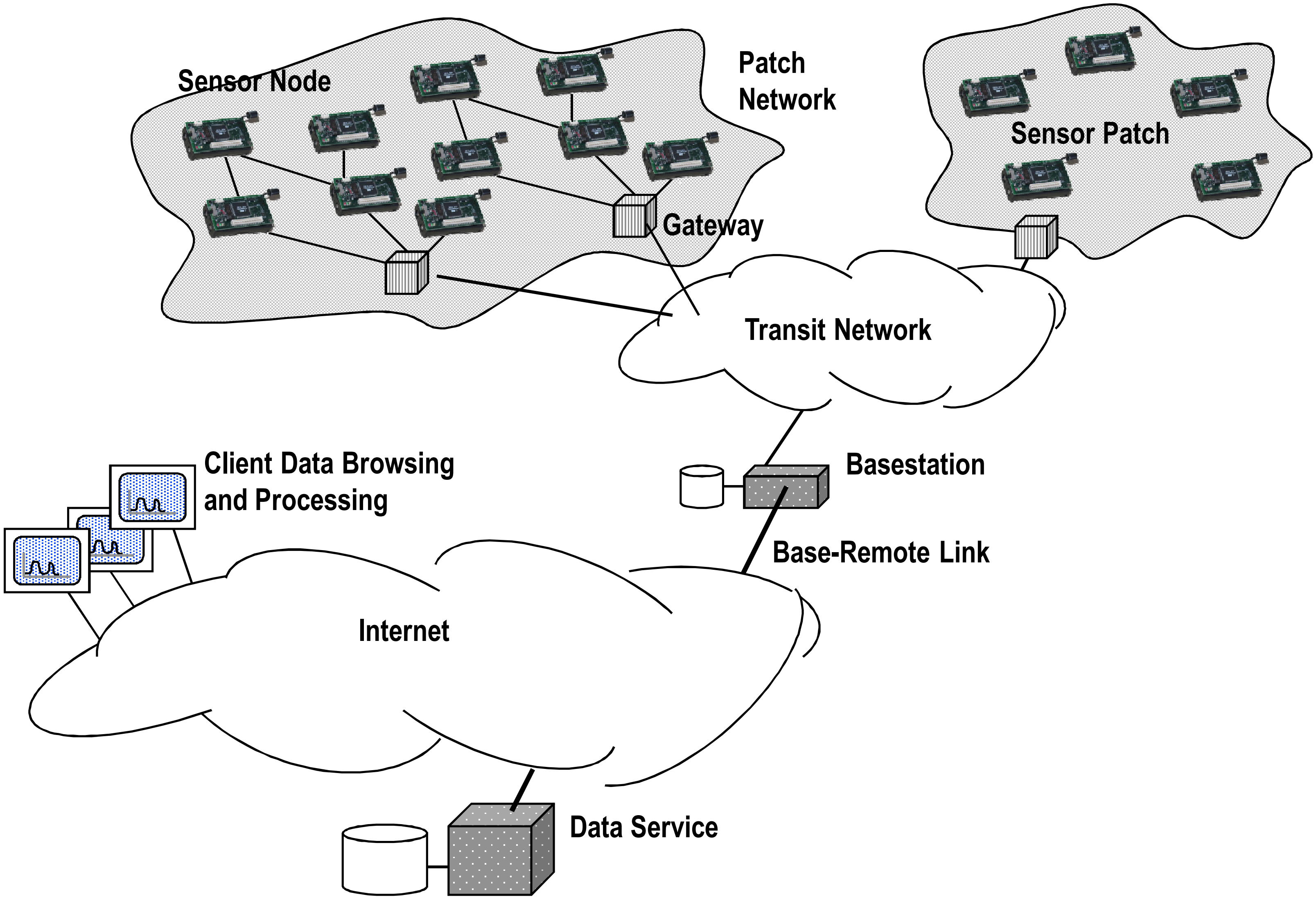
Politecnico di Milano

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

Politecnico di Milano

The architecture of a typical WSN



Politecnico di Milano

WSN: Distinctive Features

- Very large number of nodes, often in the order of thousands
  - #sensor nodes >> #users
  - Nodes need to be close to each other
    - Densities as high as 20 nodes/m3
  - Hard to rely on flooding
  - 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

Politecnico di Milano

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

Politecnico di Milano

WSNs vs. MANETs: Similarities

- Exploit wireless broadcast communication
  - And inherits all the problems, e.g., half-duplex channels, collisions, multipath effects, hidden/exposed terminal problem, …
  - Some MANET radio technologies (e.g., Bluetooth, ZigBee) are even being reused for WSN
- Communication is the most energy-intensive activity
- Bandwidth is scarce and needs to be shared
- Dynamic topology

Politecnico di Milano

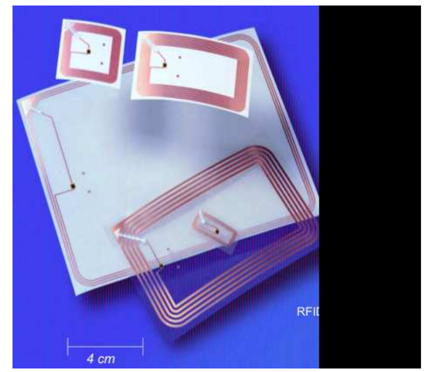
WSNs vs. MANETs: Differences

- Number and density of nodes
  - May not have global ID like IP address
  - Node failure probability
- More resource constrained
  - Sensor nodes are limited in power, computational capacities, and memory
- Nature of changes in topology
  - Except for mobile WSNs
- Goals in managing the network
  - High network lifetime vs. QoS
- Type of communication
  - Data-centric vs. point-to-point or group-based
- Role of data
  - In WSN, not all the data in the network is relevant for the user
- Role of intermediate nodes w.r.t. data
  - MANET nodes do not care about content, WSN nodes do
- MANET nodes are largely independent, WSN nodes usually are not
  - Need tight integration with sensing tasks

Politecnico di Milano

WSNs vs. RFIDs

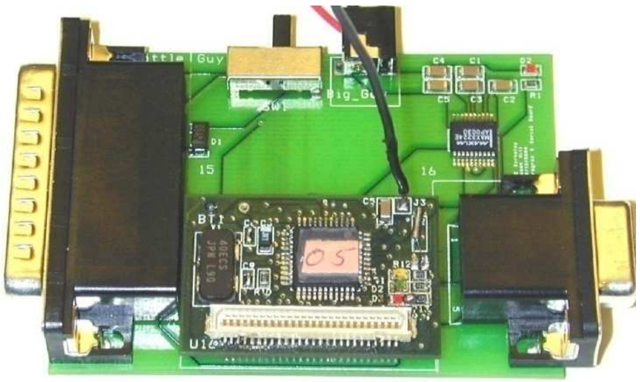
- Radio Frequency Identification
  - Used in a myriad of applications, e.g., replacement of bar codes, substitute of keys, toll paying, cattle identification, …
- Similarities:
  - Able to interact with the environment
  - Use wireless radio communication
  - Often used together
- Differences:
  - Communication technology
    - Passive RFIDs do not need an energy source, as they get it from the reader
  - Passive RFIDs are “stupid” devices
    - Active RFID have computation and communication capabilities
  - Typically used at the periphery of the system
    - Point-to-point connection to the base station
    - Rarely networked among themselves



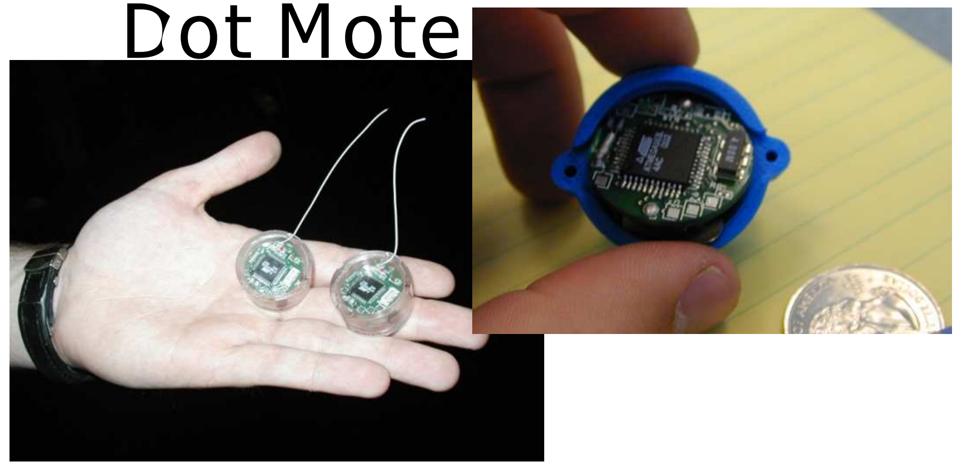
Politecnico
di Milano

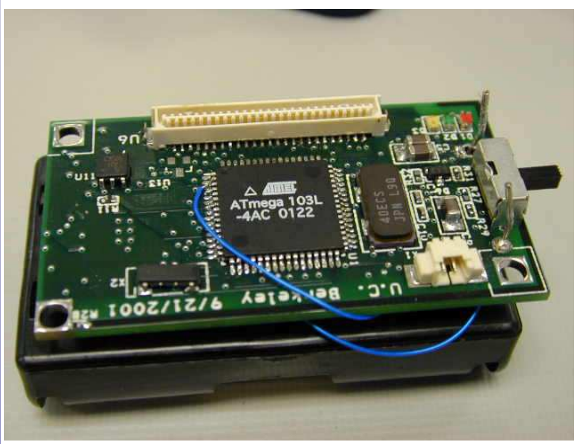
Examples of sensor nodes

Rene Mote

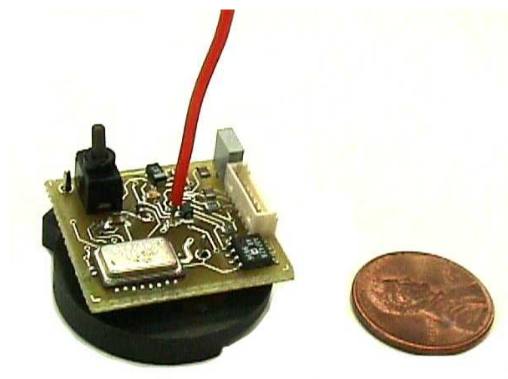


Dot Mote





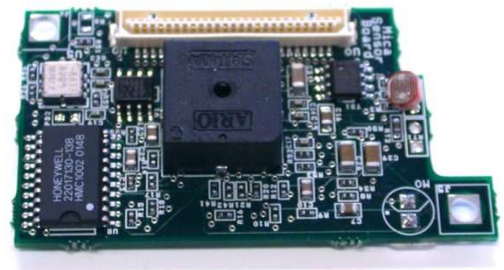
MICA Mote



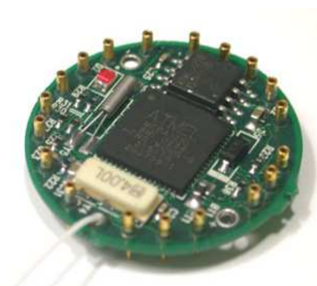
weC Mote

MICA2 Motes

- Initially developed at UC Berkeley, now commercialized by Crossbow
- Powered by two AA batteries
- Atmel ATmega128L μC
  - 8 MHz
  - 5Mbit of flash memory (128KB code, 512KB data), 4KB RAM



- Chipcon CC1000 multichannel radio
  - Range of up to 150-300m
  - Available in various frequency ranges
  - Bandwidth 38.4 kbits/s
- Modular design: different sensor boards can be mounted on the same communication board

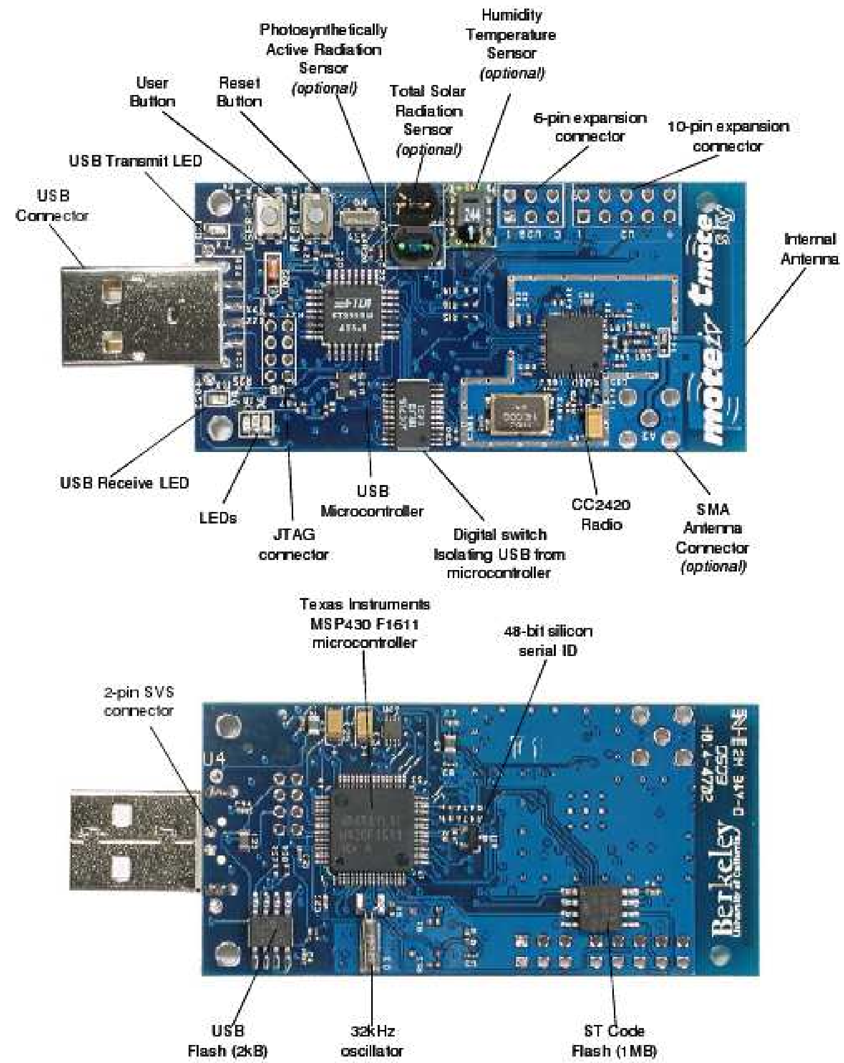


- MICA2DOT: basically same features, smaller size, fewer sensor options
- Sensors:
  - Default: acceleration, magnetic, light, temperature, acoustic
  - Available: humidity, barometric pressure, GPS module, …

TelosB/TMoteSky

Politecnico di Milano

- 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



SunSpot

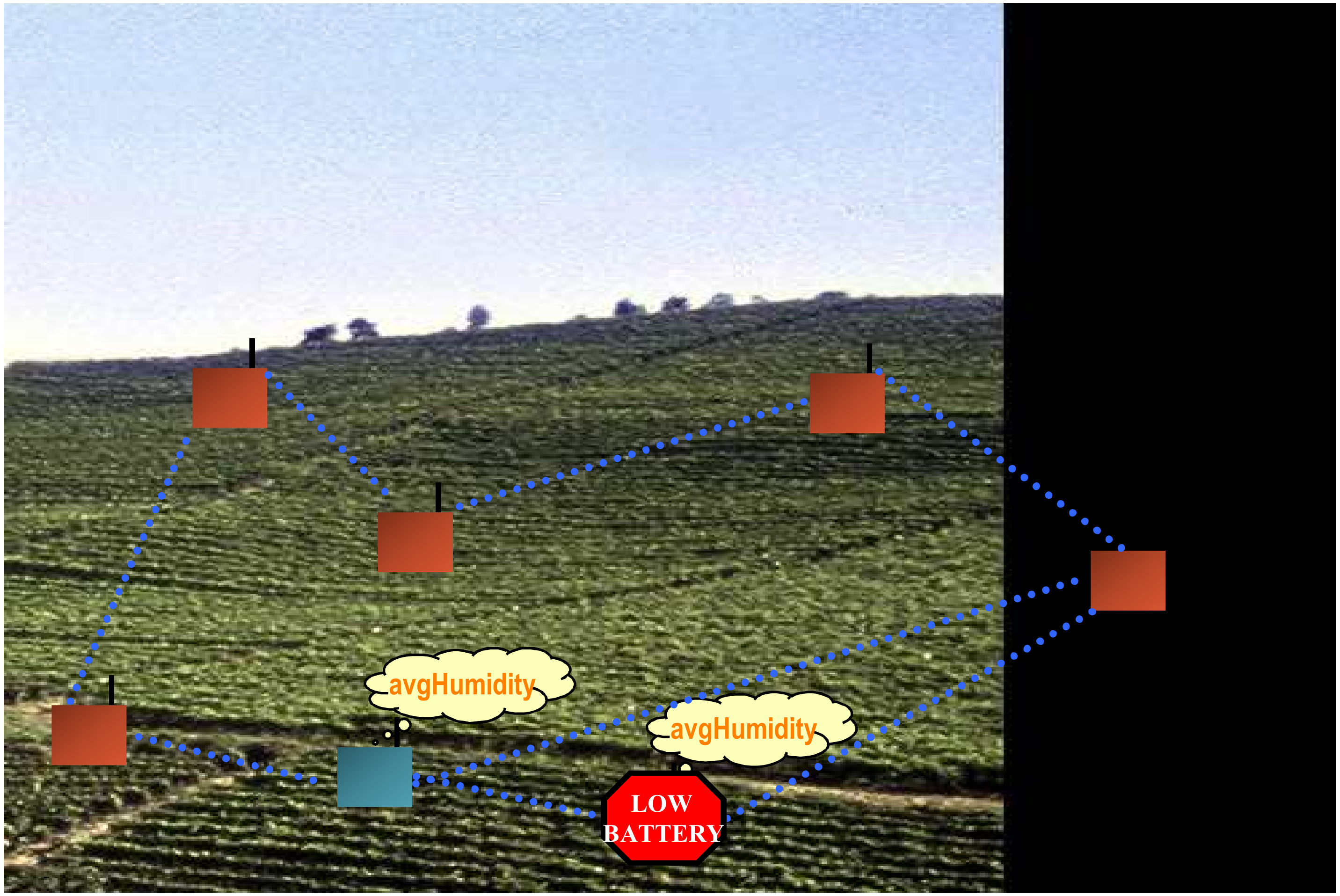
- Developed by Sun
- Built on top of the J2ME Squawk JVM, which runs directly on the processor without an OS
- Powered by a 3.7V rechargeable 750 mAh lithium-ion battery
- Atmel AT91RM9200 µC
  - 32 bit, RISC, 180 MHz core
  - 512K RAM - 4M Flash
- TI CC2420 multichannel radio
  - 802.15.4 compliant
  - ISM band (2.4 GHz)
  - Bandwidth 250 kbits/s
  - Integrated antenna: Range of up to 150m USB interface
- Sensors
  - Default: 2G/6G 3-axis accelerometer, temperature, light
  - Others: 6 analog inputs

Applications

Politecnico di Milano

- 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

Example: Controlling vineyards



Applications

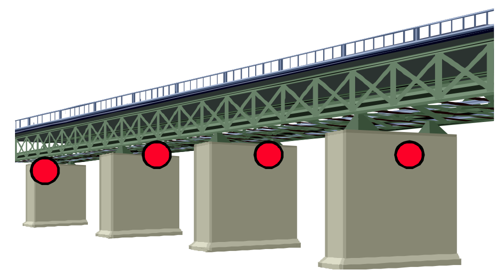
- Healthcare
  - Tracking and monitoring doctors and patients inside a hospital
  - Drug administration in hospitals
  - Diagnostics
  - Telemonitoring of human physiological data
  - Providing interfaces for the disabled
- Others
  - Factory Automation
    - Monitoring product quality
    - Machine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

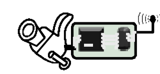
CodeBlue: WSNs for Medical Care

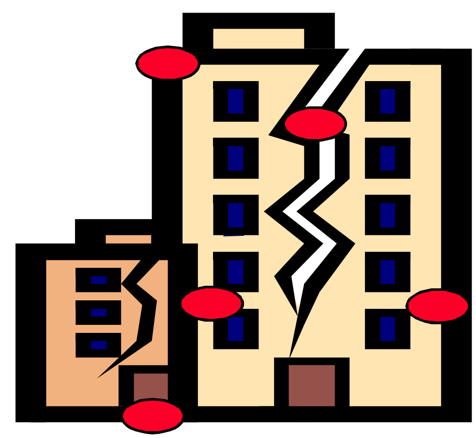
- NSF, NIH, U.S. Army, Sun Microsystems and Microsoft Corporation
- Motivation - Vital sign data poorly integrated with pre-hospital and hospital-based patient care records

Buildings (or Bridges)

- High-rise buildings self-detect structural faults
  - Reduce energy wastage by proper humidity, ventilation, air conditioning (HVAC) control
  - Needs measurements about room occupancy, temperature, air flow, …
  - Monitor mechanical stress after earthquakes

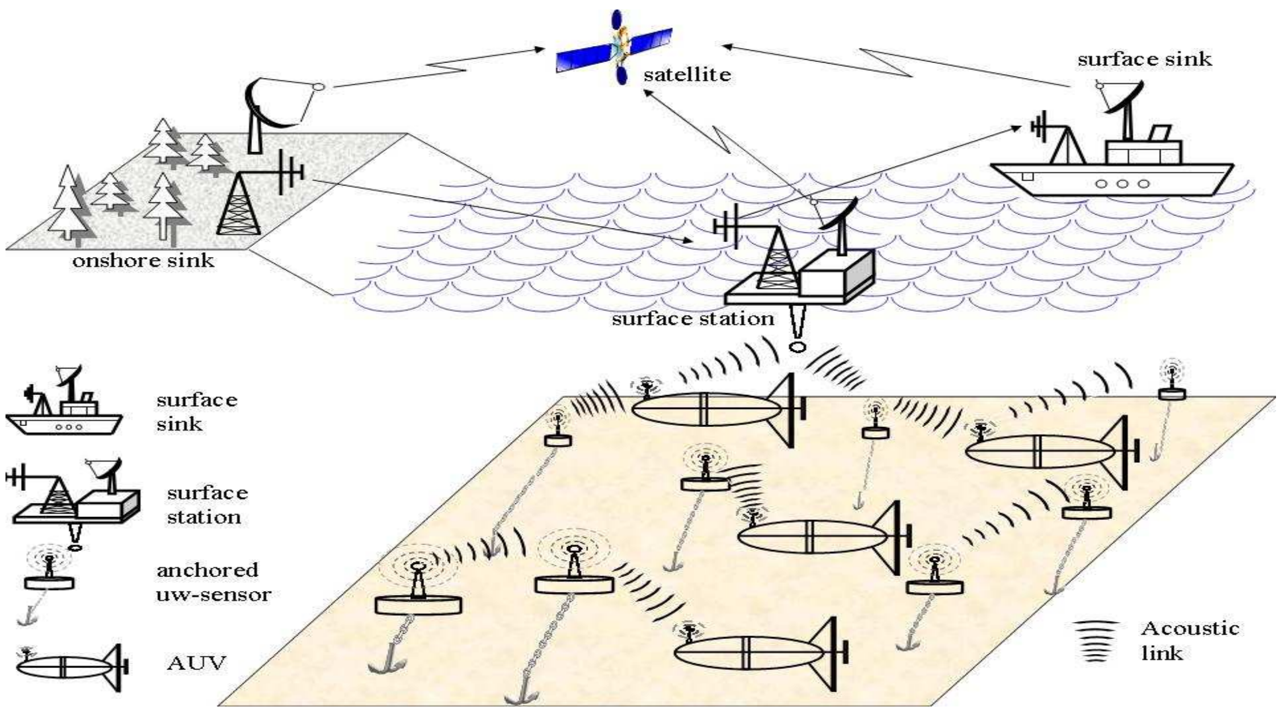






Underwater Sensor Networks

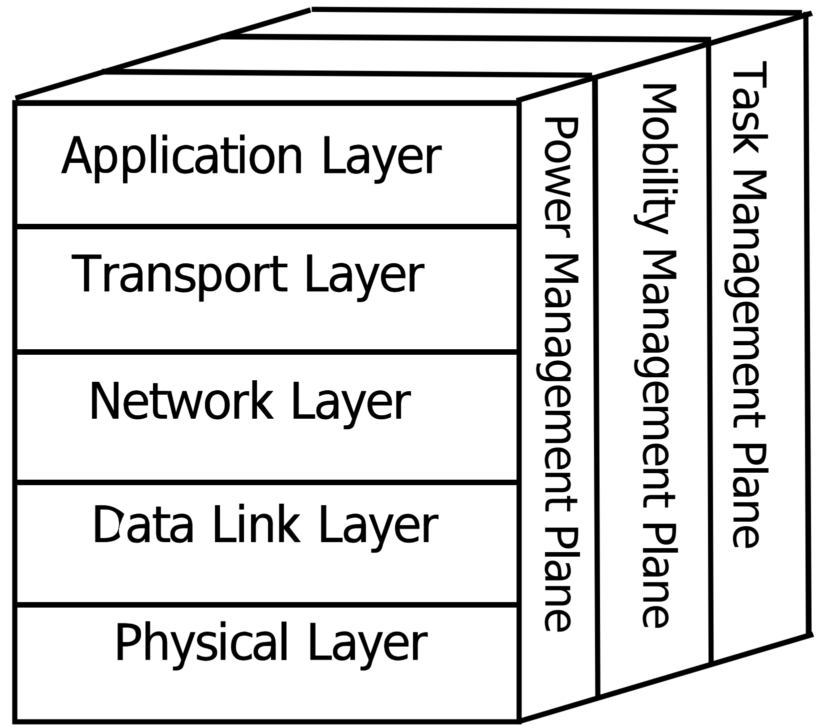
- Typically based on ultrasound
- Applications:
  - Pollution monitoring and other environmental monitoring (chemical, biological), Disaster prevention, assisted navigation, distributed tactical surveillance (mine reconnaissance)



Politecnico di Milano

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



Politecnico di Milano

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

Politecnico di Milano

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, 2004

Data-Centric Routing

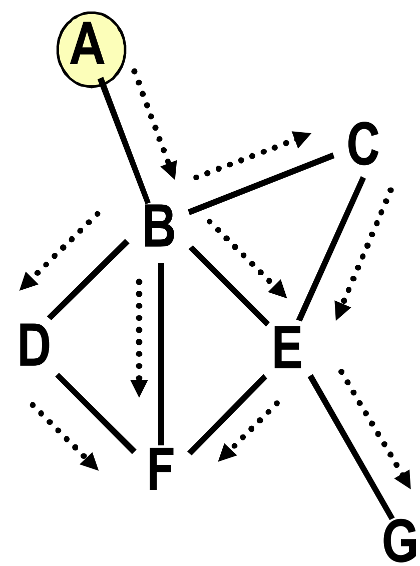
Politecnico di Milano

- 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

Politecnico di Milano

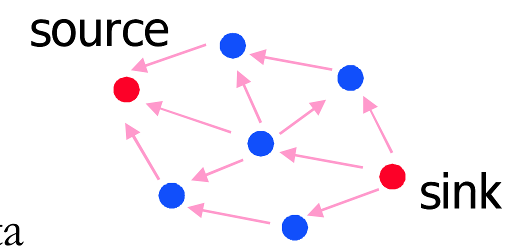
- Flooding
  - Send data to all neighbors
- Gossiping
  - Send data to one randomly selected neighbor
- Both are simple, but…
  - Implosions
    - Same data reaching the same node several times (through different routes)
  - Overlaps
    - Same data read by different nodes
  - Resource blindness (e.g.,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



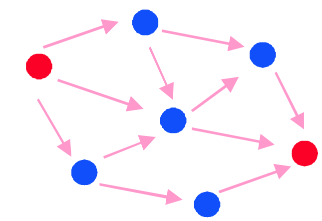
Politecnico
di Milano

Directed Diffusion

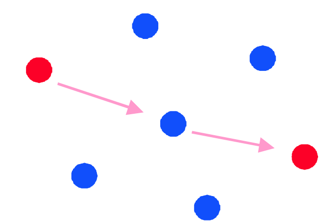
- Each sensor node names, with one or more attributes, the data it generates
- Other nodes express *interests*, based on these attributes
- Interests establish *gradients* directing the diffusion of data
  - Essentially, by setting up reverse paths
- Delivery of data matching the interest follows the route established by gradients
  - Gradients can be *reinforced* based on the quality of the data routed through them
  - Designed for *streaming* data at a given rate
- Similar to routing for content-based systems in the context of publish-subscribe
  - The notion of gradient is more sophisticated, as it accounts for the required QoS
  - Less general, as it is based on streaming



Interest Propagation



Gradient Setup



Data Delivery

TinyDB

- Idea: treat the sensor network as a database, and query it using (a modified version of) SQL
  - Hide the complexity of distribution from the end-user
- All tuples belong to a table `sensors`, which conceptually has one row per node per time instant, and as many columns as the kinds of values sensed
  - In practice, values are created on nodes only when needed, and kept only for a short period of time
- Supported by a tree overlay, which:
  - Is constructed by the query initiator
  - Must be reconstructed upon topology changes

Politecnico di Milano

TinyDB Examples

The usual SQL operators are available, although with restrictions

```
SELECT AVG(volume),room FROM sensors
 WHERE floor = 6
 GROUP BY room
 HAVING AVG(volume) > threshold
 SAMPLE PERIOD 30s
```

The result is a stream of values. A **LIFETIME** clause is also available

Materialization points store a "view" of recent data, to be used in join operations

```
CREATE
  STORAGE POINT recentlight SIZE 8
  AS (SELECT nodeid, light FROM sensors
  SAMPLE PERIOD 10s)

SELECT COUNT(*)
  FROM sensors AS s, recentLight AS rl
  WHERE rl.nodeid = s.nodeid
  AND s.light < rl.light
  SAMPLE PERIOD 10s
```

Queries can be triggered by a local system event (useful for efficient, power-saving implementation)

```
ON EVENT bird-detect(loc):
  SELECT AVG(light), AVG(temp), event.loc
  FROM sensors AS s
  WHERE dist(s.loc, event.loc) < 10m
  SAMPLE PERIOD 2 s FOR 30 s
```

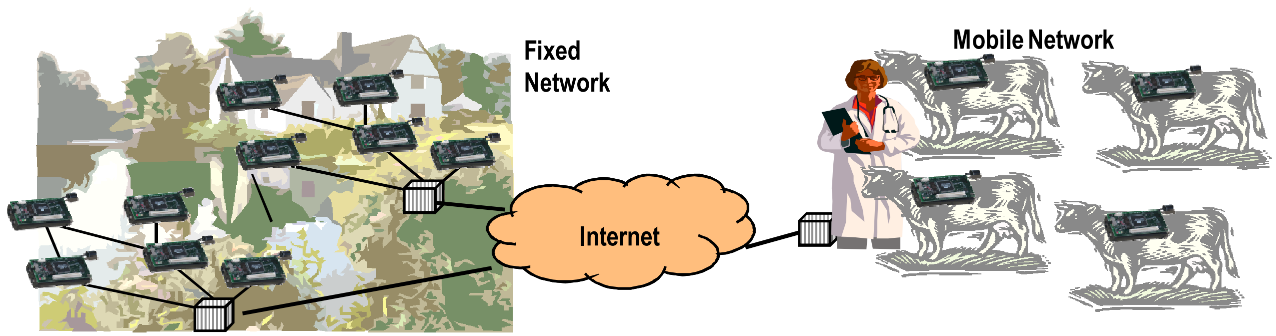
Queries can also trigger events (as well as local operations)

```
SELECT nodeid,temp
  WHERE temp > thresh
  OUTPUT ACTION SIGNAL hot(nodeid,temp)
  SAMPLE PERIOD 10s
```

Politecnico di Milano

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

Politecnico di Milano

- 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

Politecnico di Milano

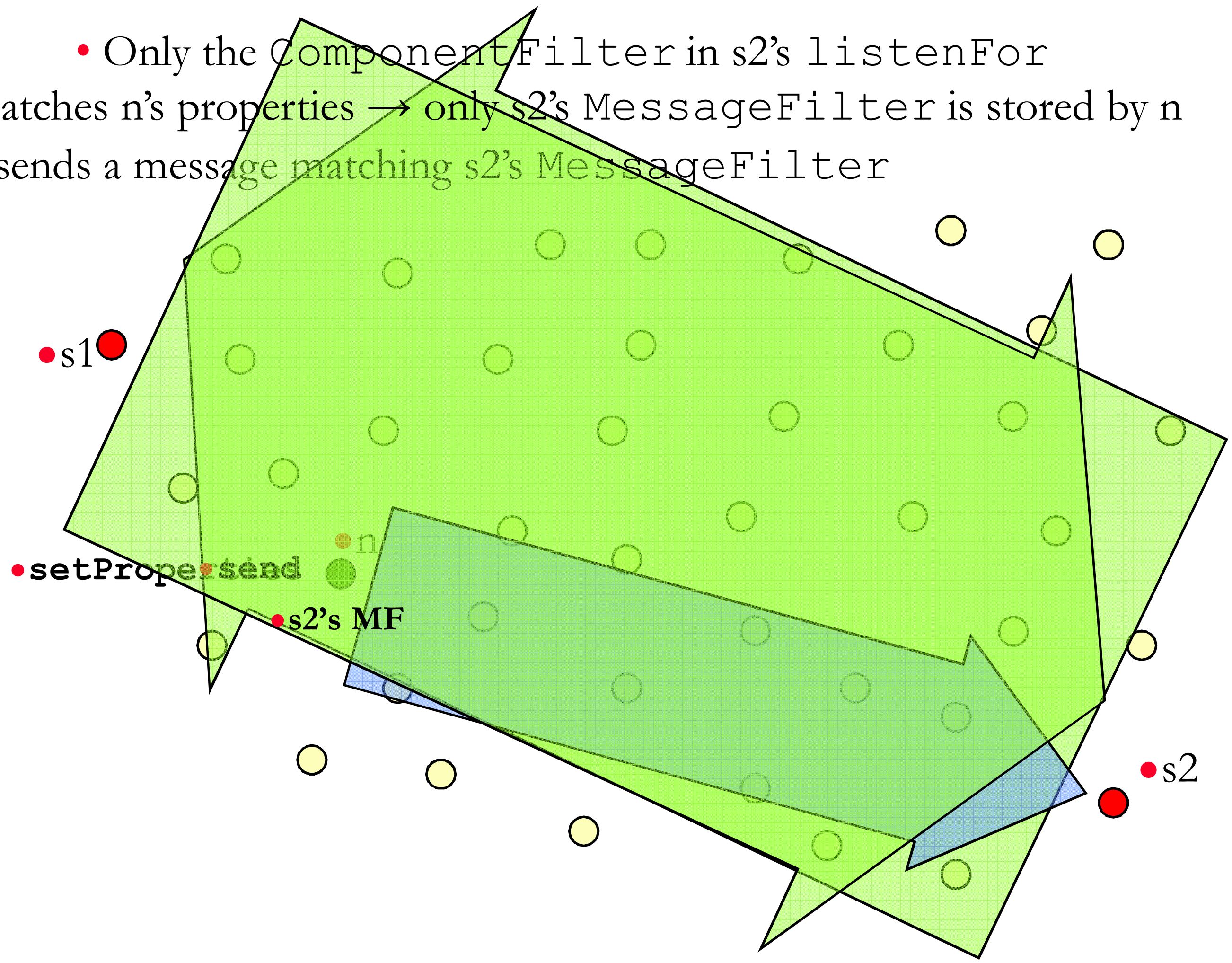
The CCBR protocol

- 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

Politecnico di Milano

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`



Politecnico di Milano

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

Politecnico di Milano

- 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

Low Energy Adaptive Clustering Hierarchy (LEACH)

Politecnico di Milano

- Idea:
  - Randomly select sensor nodes as cluster heads
  - Cluster heads can then be used kind of routers (relays) to the sink
    - Requires single hop routing, where each node can transmit directly to the cluster head and the sink
- Setup phase
  - Nodes randomly elect themselves as cluster heads
    - With a growing probability as time advances and the node has not been elected, yet
  - Cluster heads advertise their identity
  - Other nodes choose their "head" based on RSSI (to save energy)
- Steady state phase
  - Sensors begin to sense and transmit data to their cluster head which aggregates data and retransmit it to the sink
- After a certain period of time spent on the steady state, the network goes into setup phase again and enters another round of selecting cluster heads

W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-Efficient Communication Protocol for Wireless Microsensor Networks," IEEE Proc. of the Hawaii Int. Conf. on System Sciences, January 2000.

LEACH

Politecnico di Milano

- Achieves over a factor of 7 reduction in energy dissipation compared to direct communication
- The nodes die randomly and dynamic clustering increases lifetime of the system
- It is completely distributed and requires no global knowledge of the network
- It is not applicable to networks deployed in large regions
  - Assumes single-hop topology
- Furthermore, the idea of dynamic clustering brings extra overhead, e.g., head changes, advertisements etc., which may diminish the gain in energy consumption

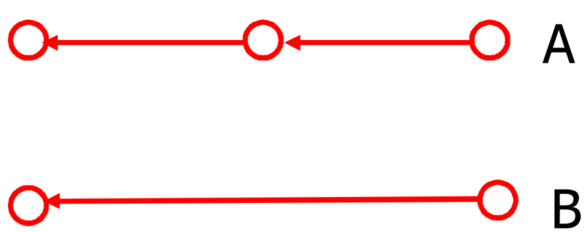
Location-Based Protocols

- 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

Minimum Energy Communication Network

Minimum Energy Communication Network (MECN), L. Li and J.Y. Halpern, "Minimum-Energy Mobile Wireless Networks Revisited" IEEE ICC 2001

Uses graph theory:
* Each node knows its exact location
* Network is represented by a graph G', and it is assumed that the resulting graph is connected
* A sub-graph G of G' is computed.
* G connects all nodes with minimum energy cost.



Connection A requires less energy than connection B because the power required to transmit between a pair of nodes increases as the n^{th} power of the distance between them ($n>=2$).

Politecnico
di Milano

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

Politecnico di Milano

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

Politecnico di Milano

S-MAC

- Aims to decrease energy consumption by sleep schedules
  - Trades off latency for reduced energy consumption
- Uses a virtual clustering of sensor nodes
  - Neighbor discovery and channel assignment combined
  - A mainly static network is assumed
- Basic Scheme: Periodic listen and Sleep
  - Choosing Schedule
    - Each node randomly chooses a time to go to sleep
    - Each node receives and follows its neighbors' schedule by setting its schedule to be the same
    - If the node receives a different schedule after it selects its own schedule, it adopts both schedules
  - Maintaining Schedule
    - Schedule is updated by sending a SYNC packet periodically

W. Ye, et. al., "Medium Access Control with Coordinated Adaptive Sleeping for Wireless Sensor Networks," IEEE/ACM Trans. on Networking, June 2004.

Politecnico di Milano

S-MAC

- Message passing
  - Only one RTS packet and one CTS packet are used
    - To avoid large control overhead and long delay
  - After the neighbor node hears the RTS and CTS, it will go to sleep for the time that is needed to transmit all the fragments (using the duration field)
  - ACK would be sent after each data fragment
    - To avoid fragment loss or error
    - To prevent hidden terminal problem
- Energy saving vs. Increasing latency (multi-hop network)
  - Carrier sense delay: Determined by the contention window size
  - Backoff delay: Because the node detects another transmission or the collision occurs
  - Transmission delay: Determined by channel bandwidth, packet length and the coding scheme adopted
  - Propagation delay: Determined by the distance between the sending and receiving nodes
  - Processing delay: Depends on the computing power of the node and the efficiency of in-network data processing algorithms
  - Queueing delay: Depends on the traffic load
  - Sleep delay: Caused by the nodes periodic sleeping

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

Politecnico di Milano

TRAMA: Energy efficient collision-free MAC

- The protocol is based on a time-slotted structure and uses a distributed election scheme based on traffic requirements of each node to determine the time slot that a node should use
- Each node gets information about its every two-hop neighbor and the traffic information of each node during a random access period, i.e., the reservation period
- Based on this information, each node calculates its priority and decides which time slot to use
- Nodes sleep during their allocated slots if they do not have any packets to send or receive

V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, ``Energy-Efficient, Collision-Free Medium Access Control for Wireless Sensor Networks,'' Proc. ACM SenSys 2003, LA, CA, Nov. 2003

Politecnico di Milano

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

Politecnico di Milano

- 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

Politecnico di Milano

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
  - Mantis
  - Contiki
  - SOS

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**

Politecnico di Milano

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

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

Declaration of provided and used interfaces

Implementation of the component's functionality

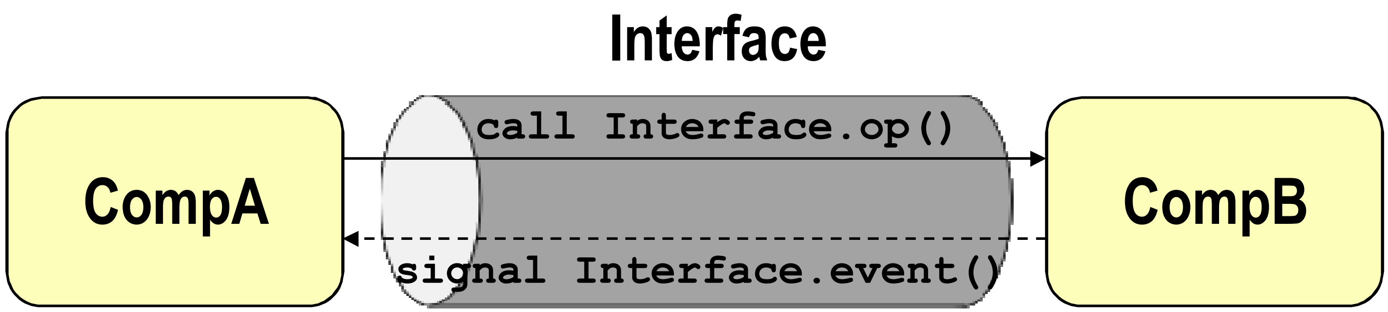
```
module FooM {
  provides {
    interface Foo;
  }
  uses {
    interface Poo;
    interface Boo;
  }
}

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

Politecnico di Milano

- 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

- 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

Politecnico di Milano

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;
    }
  }
}
```

Politecnico di Milano

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

Politecnico di Milano

- 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)

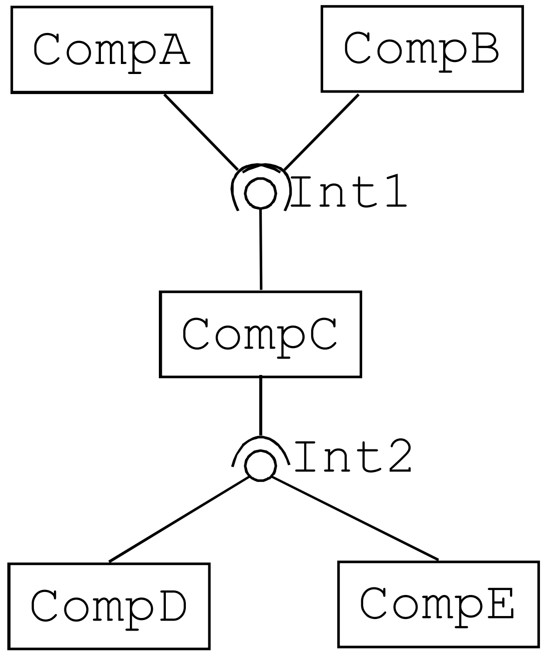
Politecnico di Milano

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;
}
```

Application setup and startup

Politecnico di Milano

- 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

Politecnico di Milano

```
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

Politecnico di Milano

```
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

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
  ```

TOSSIM

Politecnico di Milano

- 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://docs.tinyos.net/index.php/TOSSIM

Parameterized interfaces

Politecnico di Milano

- 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?

Politecnico di Milano

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

Politecnico di Milano

- 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

Politecnico
di Milano

- 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;
}
```

Politecnico di Milano

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

Politecnico di Milano

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

Politecnico di Milano

- 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

Politecnico di Milano

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

Politecnico di Milano

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

Politecnico di Milano

```
#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

Politecnico di Milano

```
#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

Politecnico di Milano

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

Politecnico di Milano

References

- TinyOs Home Page
  - www.tinyos.net
- There you will find several tutorials and the NesC reference manual