



**POLITECNICO**  
MILANO 1863



# Internet of Things Lab

## Lab 6: TinyOS (2)

# Agenda

- Using the Radio in TinyOS
- TOSSIM simulation
- Advanced Examples
- **Challenge 3**



**POLITECNICO**  
MILANO 1863

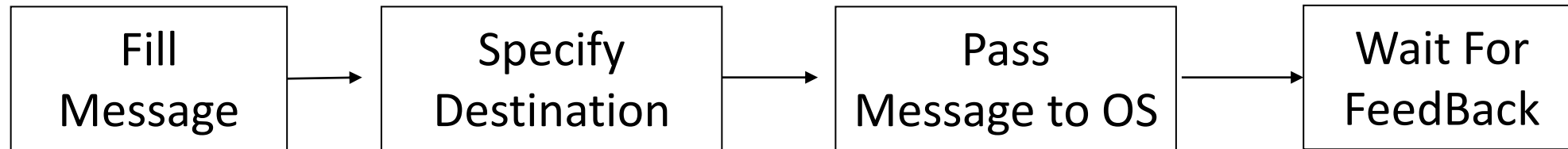


# Using the Radio

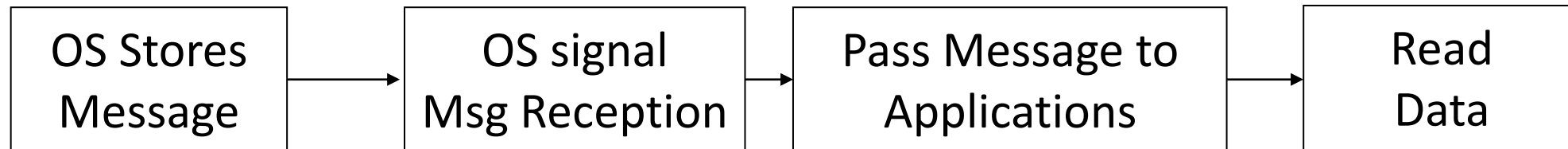
Creating/Sending/Receiving/Manipulating Messages

# General Idea

## SENDER



## RECEIVER



# Message Buffer Abstraction

- In `tos/types/messages.h`

```
typedef nx_struct message_t {  
    nx_uint8_t header[sizeof(message_header_t)];  
    nx_uint8_t data[TOSH_DATA_LENGTH];  
    nx_uint8_t footer[sizeof(message_header_t)];  
    nx_uint8_t metadata[sizeof(message_metadata_t)];  
} message_t;
```

- Header, footer, metadata: already *implemented* by the specific link layer
- Data: *handled* by the application/developer

# Message.h file

```
#ifndef FOO_H
#define FOO_H

typedef nx_struct FooMsg {
    nx_uint16_t field1;
    nx_uint16_t field2;
    ...
    nx_uint16_t field_N;
} FooMsg;

enum { FOO_OPTIONAL_CONSTANTS = 0 };

#endif
```

# Interfaces

- Components above the basic data-link layer **MUST** always access packet fields through interfaces (in /tos/interfaces/).
- Messages interfaces:
  - AMPacket/Packet: manipulate packets
  - AMSend
  - Receive
  - PacketAcknowledgements (Acks)

# Communication stack

- Active Message (AM) TinyOS communication technology
  - Include the communication protocol to send and receive packets
  - Each Packet is characterized by an AM Type:
    - Similar to a UDP port



# Communication stack

- message\_t structure:
  - Packet are represented by a message\_t structure
  - To access message\_t fields is must be used Packet and AMPacket interfaces:

```
interface Packet {  
    command uint8_t payloadLength(message_t* msg);  
    command void* getPayload(message_t* msg, uint8_t len);  
    ...  
}  
interface AMPacket {  
    command am_addr_t address();  
    command am_addr_t destination(message_t* amsg);  
    ...  
}
```

# Sender Component

## AMSenderC.nc

```
generic configuration AMSenderC(am_id_t id)
{
    provides {
        interface AMSend;
        interface Packet;
        interface AMPacket;
        interface PacketAcknowledgements as Acks;
    }
}
```

# Receiver Component

## AMReceiverC.nc

```
generic configuration AMReceiverC(am_id_t id)
{
    provides{
        interface Receive;
        interface Packet;
        interface AMPacket;
    }
}
```

# SplitControl Interface

- Radio needs to be activated prior to be used.
  - To turn on the radio, use the split-operation start/startDone
  - To turn off the radio use the split-operation stop/stopDone
- ActiveMessageC create the communication stack on evry specific architecture
  - AMSenderC component to send messages
  - AMReceiverC component to receive messages

## Example 2 – RadioCountToLeds

Create an application that counts over a timer and broadcast the counter in a wireless packet.

What do we need?

- **Header** File: to define message structure (RadioCountToLeds.h)
- **Module** component: to implement interfaces (RadioCountToLedsC.nc)
- **Configuration** component: to define the program graph, and the relationship among components (RadioCountToLedsAppC.nc)

# Message Structure

- Message structure in RadioCountToLeds.h file

```
typedef nx_struct radio_count_msg_t {  
    nx_uint16_t counter;    //counter value  
} radio_count_msg_t;
```

```
enum {  
    AM_RADIO_COUNT_MSG = 6, TIMER_PERIOD_MILLI = 250  
};
```

# Module Component

1. Specify the interfaces to be used
2. Define support variables
3. Initialize and start the radio
4. Implement the core of the application
5. Implement all the events of the used interfaces

# Module Component

- Define the interfaces to be used:

```
module RadioCountToLedsC
{
    uses interface Packet;
    uses interface AMSend;
    uses interface Receive;
    uses interface SplitControl as AMControl;
}
```

Packet Manipulation  
Interfaces



Control interface



- Define some variables:

```
implementation {
    message_t packet;
    bool locked; ...
}
```

Local Variables





# Initialize and Start the Radio

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

```
event void AMControl.startDone(error_t err) {  
    if (err == SUCCESS) {  
        call MilliTimer.startPeriodic(TIMER_PERIOD_MILLI );  
    }  
    else {  
        call AMControl.start();  
    }  
}
```

```
event void AMControl.stopDone(error_t err) { }
```

Events to report  
Interface Operation



# Implement the Application Logic

```
event void MilliTimer.fired() {  
    ...  
    if (!locked) {  
        radio_count_msg_t* rcm = (radio_count_msg_t*)call  
        Packet.getPayload(&packet, sizeof(radio_count_msg_t));  
  
        rcm->counter = counter;  
  
        if (call AMSend.send(AM_BROADCAST_ADDR, &packet,  
        sizeof(radio_count_msg_t)) == SUCCESS) {  
            locked= TRUE; }  
        }  
    }  
}
```

Create and set Packet

Send Packet

# Implement Events of Used Interfaces

```
event void AMSend.sendDone(message_t* msg, error_t error
{
    if (&packet == msg) {
        locked = FALSE;
    }
}
```

Must implement the events referred to all the interfaces of used components.

# What about Receiving?

- We need a Receive interface  
uses interface Receive;

- We need to implement an event Receive handler

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

- We need to modify the configuration component

```
implementation {
    ... components new AMReceiverC(AM_RADIO_COUNT_MSG); ... }
implementation {
    ... App.Receive -> AMReceiverC; ... }
```

# Configuration File

```
implementation {  
  
    ...  
    components ActiveMessageC;  
    components new AMSenderC(AM_RADIO_COUNT_MSG) ;  
  
    ...  
  
    App.Packet -> AMSenderC;  
    App.AMSend -> AMSenderC;  
    App.AMControl -> ActiveMessageC;  
    ...  
}
```

# RadioCountToLeds

- Let's have a look at the files (RadioCountToLeds project)
- Let's see how it works
- Let's try to turn off a device
- Can you do that in Cooja?

# Message Destination

- `AM_BROADCAST_ADDR`: for broadcast messages  
call `AMSend.send(AM_BROADCAST_ADDR, &packet, sizeof(radio_count_msg_t))`

Message will be handled by all receivers

- `mote id`: for unicast messages (e.g., 1,2 ...)  
call `AMSend.send(1, &packet, sizeof(radio_count_msg_t))`

Message will be handled only by mote 1, even if other motes are in range

# Get current mote ID

- In some cases can be useful to get the mote ID used in the simulation
- The mote ID is store in the macro **TOS\_NODE\_ID**
- e.g.: Program motes with different behaviours, based on mote ID

```
if ( TOS_NODE_ID == 1 ) {  
    //do stuff for mote 1  
}  
else if ( TOS_NODE_ID ==2) {  
    // do stuff for mote 2  
}  
else{  
    //stuff for other motes  
}
```





**POLITECNICO**  
MILANO 1863



# TinyOS SIMulator

Simulate a Wireless Sensor Network with TOSSIM

# Motivations

- WSN require large scale deployment
- Located in inaccessible places
- Apps are deployed only once during network lifetime
- Little room to re-deploy on errors

# System Evaluation

- Check correctness of application behavior
- Sensors are hard to debug!
  - “... prepare to a painful experience” [Tinyos authors’ own words]

# Simulation: Pros and Cons

- Advantages
  - Study system in controlled environment
  - Observe interactions difficult to capture live
  - Helps in design improvement
  - Cost effective alternative
- Disadvantages
  - May not represent accurate real-world results
  - Depends on modeling assumptions

# TOSSIM General Concepts

- TOSSIM is a discrete event simulator
- It uses the same code that you use to program the sensors
- There are two programming interfaces supported: Python and C++

# Key Requirements

- Scalability
  - Large deployments ( $10^3$  motes)
- Completeness
  - Cover as many interactions as possible
  - Simulate complete applications
- Fidelity
  - Capture real-world interactions
  - Reveal unexpected behavior
- Bridging
  - Between algorithm and implementation

## Example: RadioToss

- Let's simulate the RadioCountToLeds example with Tossim
- The updated code is in the folder RadioToss
- Behaviour:
  - Send a BROADCAST message with a counter
  - Turn on/off the LEDs according to the counter

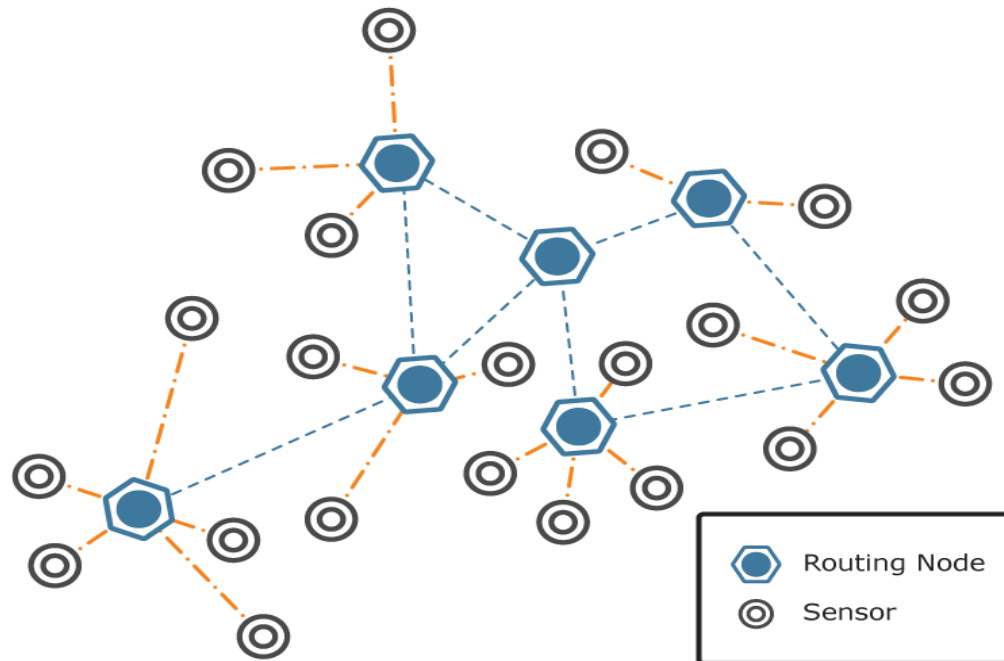
# TOSSIM Files

- TinyOS Project files:
  - RadioTossC.nc
  - RadioTossAppC.nc
  - RadioToss.h
- Topology file: topology.txt
- Noise file: meyer-heavy.txt
- Simulation script: RunSimulationScript.py



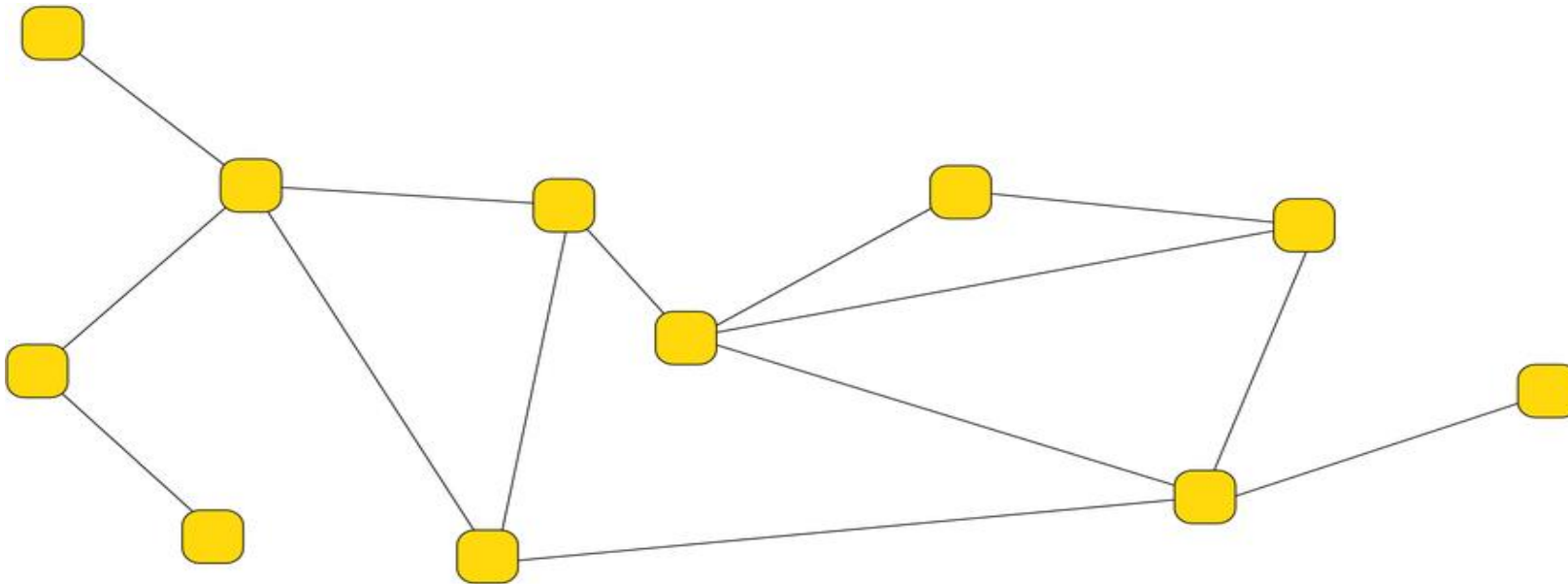
# Configuring a Network

- It's easy to simulate large networks
- You must specify a *network topology*
- The default TOSSIM radio model is signal-strength based



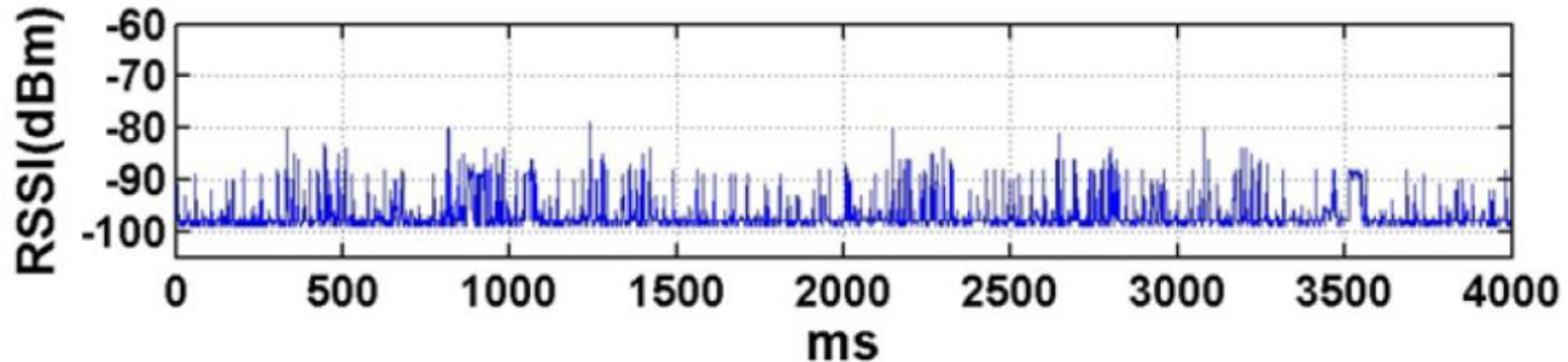
# Network Topology

- You can create network topologies in terms of channel gain:  
Each entry in the topology is formatted as  
source destination gain *i.e.* 1 2 -54.0



# Radio Channel

- You need to feed a noise trace (meyer-heavy.txt file)
- The directory `tos/lib/tossim/noise` contains some sample models
- On reception, SNR is evaluated for each bit of the message



# Debugging Statements

- The output is configurable by debug channels
  - `<ch>` identifies the output channel
  - `<text>` text debugging and application variables
- `dbg(<ch>,<text>)` → **DEBUG(ID):<text>**
- `dbg_clear(<ch>,<text>)` → **<text>**
- `dbgerror(<ch>,<text>)` → **ERROR(ID):<text>**
- A channel can have multiple outputs

# How to Run TOSSIM?

- To compile TOSSIM you pass the sim option to make:  
**make micaz sim**
- To run TOSSIM use the RunSimulationScript. It must be in the same folder of the TinyOS files

**python RunSimulationScript.py**

# Simulation

- Setup Debug Channels

```
#Add debug channel
print "Activate debug message on channel init"
t.addChannel("init",out);
print "Activate debug message on channel boot"
t.addChannel("RadioCountToLedsC",out);
print "Activate debug message on channel radio"
t.addChannel("radio",out);
print "Activate debug message on channel radio_send"
t.addChannel("radio_send",out);
print "Activate debug message on channel radio_ack"
t.addChannel("radio_ack",out);
print "Activate debug message on channel radio_rec"
t.addChannel("radio_rec",out);
print "Activate debug message on channel radio_pack"
t.addChannel("radio_pack",out);
print "Activate debug message on channel role"
t.addChannel("role",out);
```

- Number of events: increase/decrease according to your simulation

```
107 for i in range(0,1200):
108     t.runNextEvent()
109
```

# Attention on RunSimulationScript

Adapt the script for the correct number of nodes (after creating them)

- Add noise trace for all nodes:

```
print "Reading noise model data file:", modelfile;
print "Loading:",
for line in lines:
    str = line.strip()
    if (str != "") and ( compl < 10000 ):
        val = int(str)
        mid_compl = mid_compl + 1;
        if ( mid_compl > 5000 ):
            compl = compl + mid_compl;
            mid_compl = 0;
            sys.stdout.write("#")
            sys.stdout.flush()
        for i in range(1, 3):
            t.getNode(i).addNoiseTraceReading(val)
print "Done!";
```

- Create noise model for all nodes

```
for i in range(1, 3):
    print ">>>Creating noise model for node:", i;
    t.getNode(i).createNoiseModel()
```

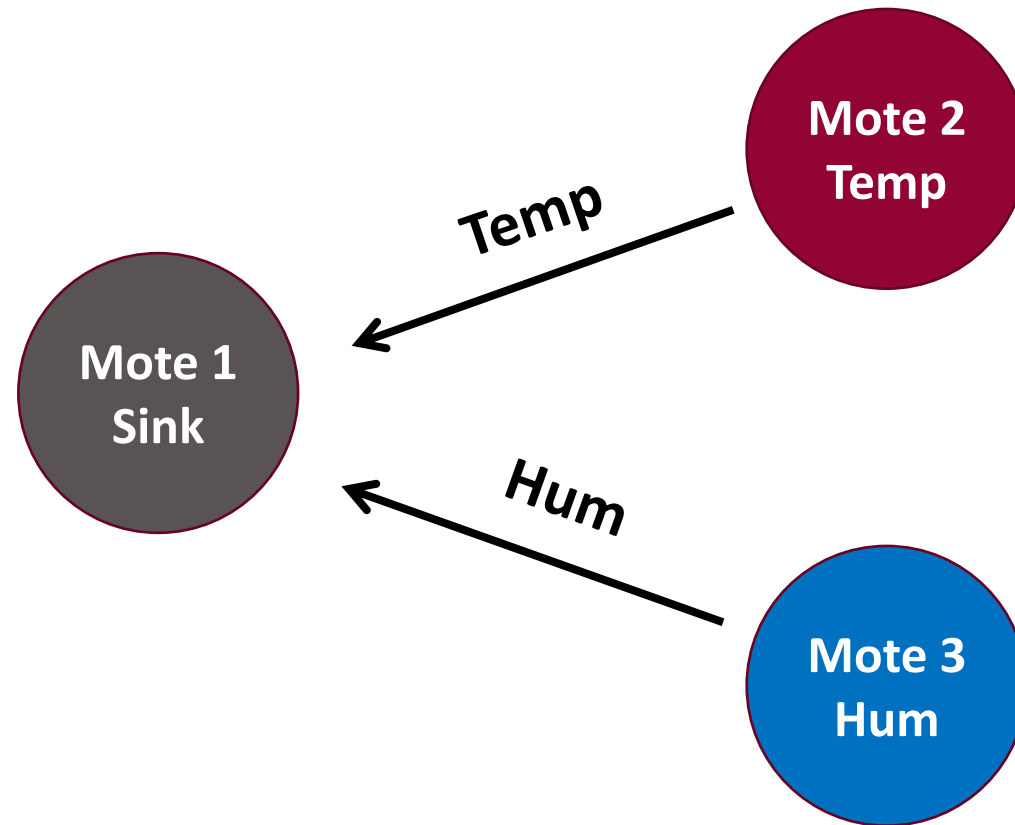
Range(1,n\_nodes+1)

# Exercise: temp/hum sensor

- Starting from the example in:  
IOT-examples/TinyOS/Supporting\ Files/TempHumSensor
- Simulate with TOSSIM a TinyOS application that:
  - Generate 3 motes (1, 2, 3)
  - Mote #1 is the sink
  - Mote #2 is the temperature sensor
  - Mote #3 is the humidity sensor



# Topology



# Fake temp/hum sensor

- Mote #1 only receives messages from #2 and #3
- Mote #2 sends periodic (every 1 second) messages to the sink (#1)
- Mote #3 sends periodic (every 2 seconds) messages to the sink (#1)

# Message format

- Messages are composed of the following fields:
  - **Type:** 0 or 1 (0 for temperature, 1 for humidity)
  - **Data:** the value of the sensor