Politecnico di Milano

Advanced Network Technologies Laboratory

Using the Radio

Creating/Sending/Receiving/Manipulating Messages

General Idea

SENDER

Fill Message → Specify Destination → Pass Message to OS → Wait For FeedBack

RECEIVER

OS Stores Message → OS signal Msg Reception → Pass Message to Applications → Read Data

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

```
#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
```

Header and Metadata

```
typedef nx_struct cc2420_header_t {
    nxle_uint8_t length;
    nxle_uint16_t fcf;
    nxle_uint8_t dsn;
    nxle_uint16_t destpan;
    nxle_uint16_t dest;
    nxle_uint16_t src;
    nxle_uint8_t network; // optionally included with 6LowPAN
    layer
    nxle_uint8_t type;
} cc2420_header_t;

typedef nx_struct cc2420_metadata_t {
    nx_uint8_t tx_power;
    nx_uint8_t rssi;
    nx_uint8_t lqi;
    nx_bool crc;
    nx_bool ack;
    nx_uint16_t time;
} cc2420_metadata_t;
```

Interfaces

- Components above the basic data-link layer MUST always access packet fields through interfaces (in /tos/interfaces/).

- Messages interfaces:
  - AMPacket: manipulate packets
  - AMSend
  - Receive
  - PacketAcknowledgements (Acks)

Sender Component

- AMSenderC

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

Receiver Component

- AMReceiverC

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

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; }
    }
}
```

Creates and Set Packet

Send Packet

Implement Events of Used Interfaces

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

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

And 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 - Demo

- Let’s have a look at the files
- 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))
```

- mote id: for unicast messages (e.g., 1)

```
call AMSend.send(1, &packet,
sizeof(radio_count_msg_t))
```

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.: check if we are on mote 1:

```
if ( TOS_NODE_ID == 1 ) {
   //do something
   }
```

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

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

RadioTossim

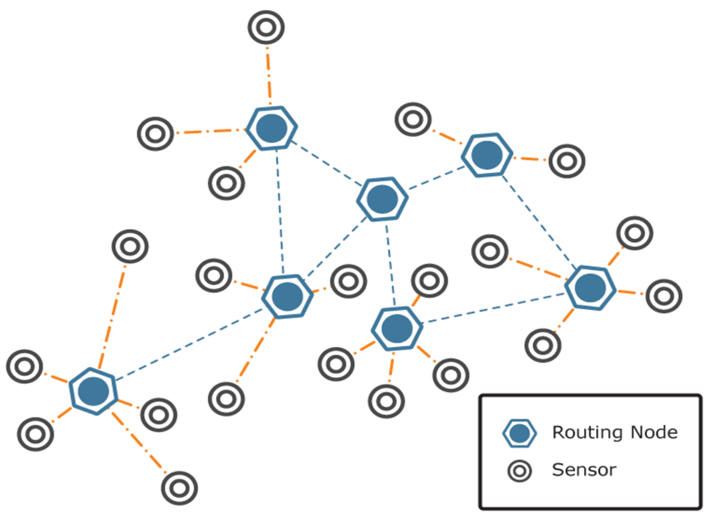
- Let’s simulate the RadioCountToLeds example with Tossim
- The updated code is in the folder <u>RadioToss</u>
- Behaviour:
  - Send a BROADCAST message with a counter
  - Turn on/off the LEDs according to the counter

TOSSIM Files

- Let's use RadioToss example
  - TinyOS files:
    - `RadioTossC.nc`
    - `RadioTossAppC.nc`
    - `RadioToss.h`
  - Topology file: `topology.txt`
  - Noise file: `meyer-heavy.txt`
  - Simulation script: `RunSimulationScript.py`

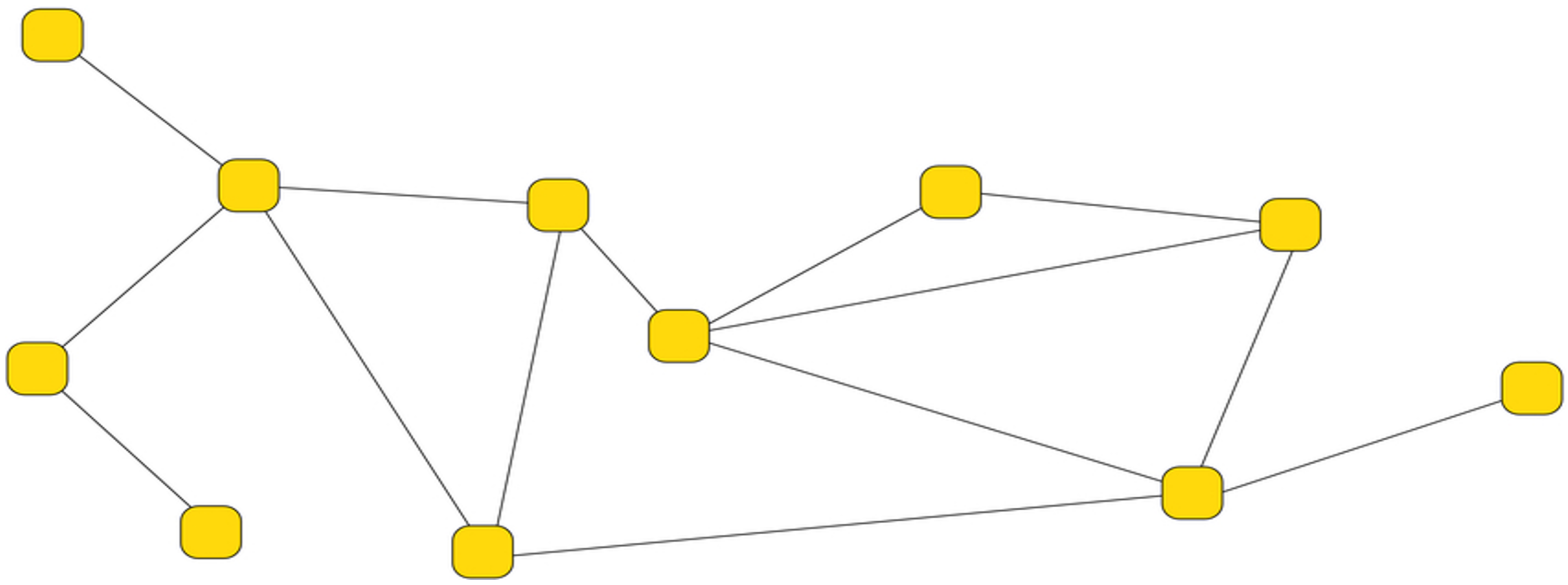
Configuring a Network

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



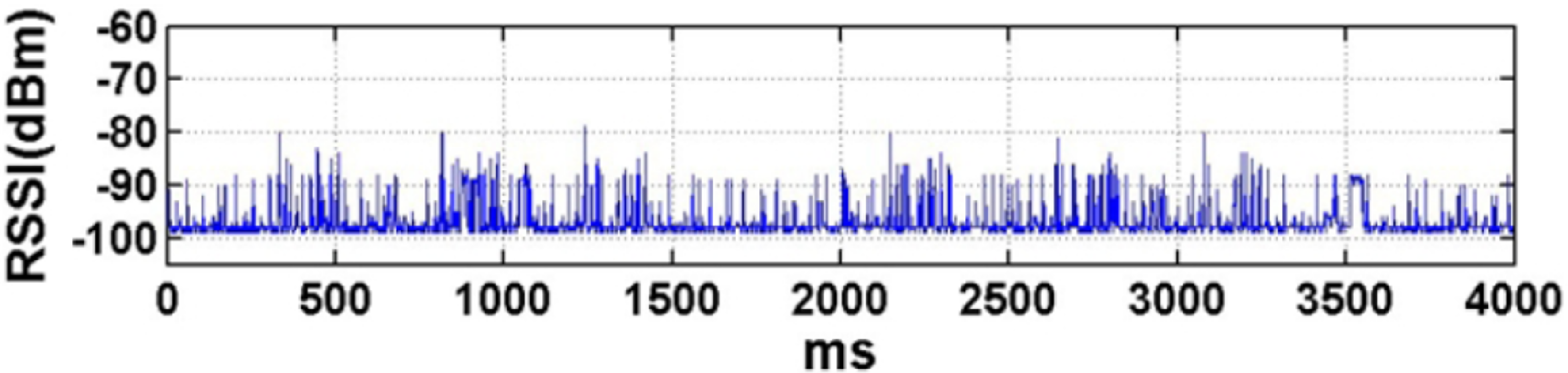
Network Topology

- You can create network topologies in terms of channel gain
  - source, the destination and the gain (for example “1 2 -54.0”)



Radio Channel

- You need to feed a noise trace
- 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 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

- Debug channel

```
#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

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

Exercise: temp/hum sensor

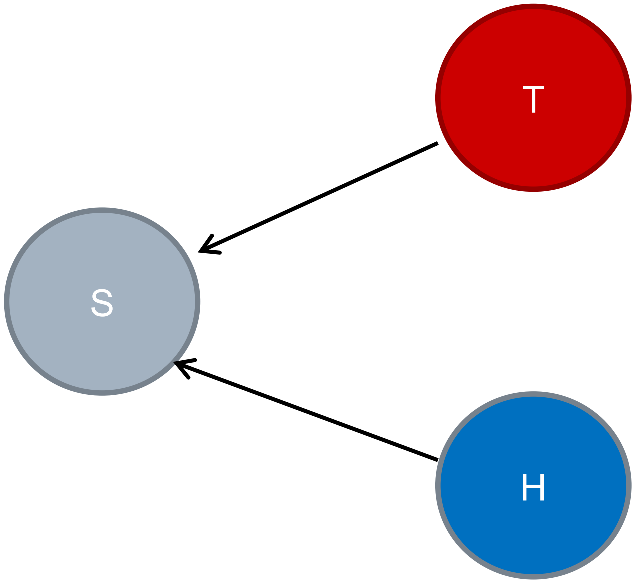
- Starting from the example in:

`IOT-examples/TinyOS/Supporting\ Files/TempHumSensor`

Simulate with TOSSIM a TinyOS application that:

- generate 3 motes (0, 1, 2)
- mote #0 is the sink
- mote #1 is the temperature sensor
- mote #2 is the humidity sensor

Topology



Fake temp/hum sensor

- Mote #0 only receive messages from #1 and #2
- Mote #1 sends periodic (every 1 second) messages to the sink (#0)
- Mote #2 sends periodic (every 2 seconds) messages to the sink (#0)

Messages

- Messages are composed like this:

```
- type: 0 or 1 (0 for temp and 1
  for hum)

- data: the value from the fake
sensor
```