Intro to TinyOS

Daniele Puccinelli

daniele.puccinelli@supsi.ch

http://web.dti.supsi.ch/~puccinelli

What is TinyOS?

- TinyOS: an operating system for resource-constrained devices
- It offers you the tools to use the available features of your hardware
- Not exclusive to sensor networks, though widely used for them
- No clear separation between user and programmer
- Written in nesC, a C dialect

Why do we need it?

Having to program all the components of the sensor nodes from scratch would be a daunting task

- Parallel distributed programming...
- ...on resource-constrained devices

- There are things you need to use but don’t care about

Reuse other people’s code

- There are things you do care about

Focus on those, do them well, and let people use them

Basics of nesC

nesC = network embedded system C
component-based C dialect

nesC components have a local namespace

- component A calling function *f*: A.*f* is introduced into the global namespace

- component Z calling function *f*: Z.*f* is introduced into the global namespace

- A.*f* and Z.*f* may be entirely different

Provide/Use: Example

```
module Sense {
provides command measure;
uses command filter;}
```

- Sense **provides** a *measure* tool to its fellow modules
- Sense must define how that works

- Sense **uses** a filter to smooth out its measurements
- Sense gets that for free and need not define it

Provide/Use: More in General

```
module A{
provides command use_A_to_do_f;
uses command use_B_to_do_g;}
```

- A knows how to do *f* (Sense knows how to measure)
- B knows how to do *g* (Filter knows how to smooth out a signal)

- A provides a command for others to do *f*
- Others will use A to do *f* A’s way
- A uses B to do *g* B’s way

Events

Events are the generalization of interrupts

Command: drive something
Event: get driven by something

- Your radio module signals that a packet was received...
- ...or that a packet just got sent
- Your timer signals that a certain amount of time has elapsed
- Your sensing module signals that the sample is ready
- Your low-pass filter signals that the sample mean is zero

Interfaces

Interfaces are sets of related functions in the form of header files
An interface is a list of all you can do on a given theme

```
Interface StdControl{
command start();
command stop();}

Interface Radio{
command sendPacket(packet);
command measureSignalStrength();
command dutyCycle();  // start/stop at a higher level
event packetReceived();
}
```

Modules and Interfaces

```
module TemperatureSensor
{provides command measure(sampling_time);
 uses command filter();}

module LightSensor
{provides command measure(sampling_time);
 uses command filter();}

interface Sense
{command measure(sampling_time);
command filter();}
```

Reusing Modules through Interfaces

module DoubleSenseC
{uses Sense as senseTemp;
uses Sense as senseLight;}

DoubleSenseC leverages modules xSensor to sense x
The interface is the same across different sensors

module A{
provides interface do_*f*;
uses interface do_*g*;}

Many modules can do_*f* and/or do_*g* in different ways

Wiring

The process of connecting users and providers
Done in **Configuration** files

```
configuration DoubleSenseAppC
{}

implementation
{
components TemperatureSensor as T;
components LightSensor as L;
components SenseC as S;

S.senseTemp -> T;
S.senseLight -> L;

}
```

Summary

Code is broken up into **components**
(discrete units of functionality)

Components can **use** functions defined by others...
...and **provide** functions to others

Compile-time composition: no dynamic loading of new stuff

○ Bad idea for user-driven systems (like your computer)

○ Great for embedded systems
• untethered operation (if you are not there...)
• faults are deadly (...who reboots your mote?)