IOT EMULATION WITH COOJA

BA BAGULA & ZENVILLE ERASMUS
ISAT LABORATORY
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF THE WESTERN CAPE (UWC)
CAPE TOWN – SOUTH AFRICA

ICTP-IoT Workshop, Trieste 16-27 March 2015
Outline

- What is Cooja
  - Emulator vs Simulator
  - Main steps
  - Hello-world
- More examples
  - UDP-RPL/broadcast
  - Z1 sensors
  - Sense/Send/Blink
  - UDP-RPL/Unicast
- Energy monitoring
  - Timeline
  - Energest
  - Powertrace
  - PowerTracker
- Networking protocols
  - RPL
  - LIBP
  - Multi-sink
Cooja is an emulator

- According to different sources, an emulator is:
  - a hardware or software system that enables one computer system (called the host) to **behave like another** computer system (called the guest): e.g. Cooja enabling your laptop to behave like a Z1 mote.
  - a system that typically **enables** the host system to **run software or use peripheral devices** designed for the guest system: e.g. Cooja enabling your laptop to run the RPL protocol, LIBP and/or other IoT protocols of interest.
Cooja is an emulator

- According to different sources, an emulator is:
  - a system that *behaves exactly like* the guest system, and abides by all of the rules of the system being emulated, but operating in a different environment to the environment of the original emulated system.
  - a *complete replication* of the guest system, right down to being binary compatible with the emulated system's inputs and outputs.
Cooja is not a simulator

- According to different sources, a simulator is:
  - a hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest), but is implemented in an entirely different way: e.g. A flight simulator gives you the feeling of flying an airplane, but you are completely disconnected from the reality of flying the plane, and you can bend or break those rules as you see fit. e.g. Fly an Airbus A380 upside down between London and Sydney without breaking it.
According to different sources, a simulator is:

- a system that *provides the basic behaviour* of a system *but may not necessarily abide by all of the rules* of the system being simulated.

- A system designed to *recreate the operation or behaviour* of the guest system. The underlying principles can be the same as the original or different.
What is Cooja?

- Cooja is a Contiki network emulator
  - An extensible Java-based simulator capable of emulating Tmote Sky (and other) nodes
- The code to be executed by the node is the exact same firmware you may upload to physical nodes
- Allows large and small networks of motes to be simulated
- Motes can be emulated at the hardware level
  - Slower but allows for precise inspection of system behaviour
- Motes can also be emulated at a less detailed level
  - Faster and allows simulation of larger networks
Cooja (continued)

- Cooja is a highly useful tool for Contiki development
  - It allows developers to test their code and systems long before running it on the target hardware
  - Developers regularly set up new simulations to
    - debug their software
    - to verify the behaviour of their systems
Main steps

1. Open a terminal window to start Cooja
2. Create a new simulation to run Contiki in simulation and wait for Cooja to start and compile itself
3. Set simulation options
4. Create a new mote type
5. Add motes to the simulation
6. Open a terminal Cooja is a highly useful tool for Contiki development
   1. It allows developers to test their code and systems long before running it on the target hardware
   2. Developers regularly set up new simulations to debug their software and verify the behavior of their systems.
1. Starting Cooja

- Open a terminal window

To start Cooja, first open a terminal window.

- cd contiki/tools/cooja (Cooja directory)
- start cooja by issuing ant run
Waiting for Cooja to start

- When Cooja first starts, it will compile itself. This may take some time.

When Cooja is compiled, it will start with a blue empty window.
2. Create a new simulation

- Click the **File** menu and click **New simulation...**
3. Set simulation options

- Cooja now opens up the **Create new simulation** dialog. Either change the dialog name or stick with *My simulation*.

- Click the **Create** button.
Cooja brings up the new simulation.

- Shows all the motes
- Shows all serial port printouts from all the motes
- Shows all communication events over time
4. Add motes to the simulation

- Add motes
Create a new mote type

- Cooja opens up the **Create Mote Type** dialog

- choose a name for our mote type
- choose the Contiki application that our mote type will run
5. Find Contiki Application

- Hello World /opt/contiki-2.7/examples/hello-world

- Specify application C source file → Open
6. Compile the Contiki application

- Cooja will verify that the selected Contiki application compiles for the platform that we have selected

- Click on the Compile button. This will take some time...

- Compilation output will show up in the bottom white panel.
7. Create the mote type

- Click on the **Create** button to create the mote type. The window will close.

![Create Mote Type: Compile Contiki for sky](image)
8. Add motes to simulation

- Add motes by changing the number of motes in the **Number of motes** field to 5.

![Add motes (Sky Mote Type #sky1) dialog box]

- Click on **Add motes** to add motes to the simulation.
9. Start the simulation

- The 5 added motes are now seen in the simulation window.

- Click the **Start** button to start the simulation.
10. Pause the simulation

✓ View → Select Log output: printf()'s
11. Get some statistics

- Mote output window
  - Printouts from the simulated motes
- Network window
  - Shows ongoing network communication
- Timeline
  - Shows communication and radio events over time
  - The small gray lines are ContikiMAC periodically waking up the radio
- Pause
  - Click the **Pause** button to pause the simulation
More examples

- simple-udp-rpl/broadcast-example.c
More examples

- Z1 sensors
More examples

- Sense, send and blink with receive and blink
More examples

- unicast-example.c
  - ipv6
  - simple-udp-rpl
Timeline in COOJA

- Radio ON/OFF
  - No colour: radio off
  - Grey: radio on

- Radio RX/TX
  - Green: received a packet
  - Blue: packet sent
  - Red: interfered radio (collisions etc.)

- Right-clicking will reveal additional info.[2]
Measure Power Consumption with Energest

- Can be used for obtaining per-component power consumption on Contiki.
  - (cpu_ON, LPM, TX, RX)
  - i.e. the time the radio was in RX mode (rxon)
- For RX:
  - Power(mW) = (rxend – rxstart) * 20mA * 3V / 4096 / runtime(seconds)
  - If you do not divide by runtime you get the energy consumption during runtime.
Measure Power Consumption: Powertrace

- Uses Energest along with a periodic difference of the rtimer ticks to get average power over a shorter period of time or for particular network modes[3].
- Periodically prints out power consumption
Measure Power Consumption: PowerTracker

- A COOJA plugin that measures the average simulated radio duty cycles.

  simple-udp-rpl/broadcast-example.c
Network Protocols

- COOJA has 2 stacks: uIP and Rime
- Protocol stacks may be interconnected
  - uIP data can be transmitted over Rime and vice versa
- Cooja can be used to emulate network protocols:
  - RPL
  - LIBP
LIBP, known as the **Least Interference Beaconing Protocol**, is the implementation of the **Least Interference Beaconing Algorithm**, LIBA.

LIBP extends the beaconing process widely used by collection protocols with load balancing to improve the Ubiquitous Sensor Network (USN) energy efficiency\[4\].

The process involving the least interference paradigm allows the selection of a parent node that has the **smallest number of children**. This is a point of **least traffic flow interference**.

The parent selection model chooses the first parent node heard from, whereby the sensor nodes hear from a set of neighbours and select the least burdened (in number of children) as the parent node.
Upon network startup, Rime started with address 8.0 for Node ID 8.
The image details the radio channel as 26 and the channel check rate of 8 Hz.
Sink mote with ID 1 – After 2 minutes, 0 seconds and 673 milliseconds, ID 1 broad-casted to the network that it is sink by sending “Hi from sink thread”.
PowerTracker after 5 minutes.

Sky 3 used the most power by being on most of the time. It's Radio TX is also the highest with a value of 0.73%. It has the second highest Radio RX of 0.09%.
Libp – Parent with children

Node 3 has 2 children, namely nodes 5 and 6. These nodes also have children.

Sky 10 used the least power with its Radio on at 1.73%
Radio TX at 0.47% and
Radio RX of 0.02% (least percentage).

Sky 10 is ranked along the bottom of the tree, has no children and is only active when it has to send its data, unlike the other motes.
LIBP – large network

- Starting COOJA with “ant run” will give you the default Java maximum memory
  - 5 – 10 emulated nodes
- If you use “ant run_bigmem” you will be able to simulate/emulate larger networks.
Compiled **50 Cooja Motes**
(Simulated motes – run as native java code)
Downside: Cannot do any power profiling
LIBP – 2 sinks

Node ID 1 : Primary Sink
Node ID 2 : Secondary Sink

Load balanced network with 2 sink nodes
LIBP – 2 sinks

- The 2 sink nodes help with load balancing and network recovery
  - e.g. when one node goes offline, it's children (orphaned nodes) attempt to connect to the other sink

Secondary sink - offline  
Recovery in process
LIBP – 2 sinks

Recovery in process

Orphaned nodes also attempt to connect with each other

Network has recovered and every node is making use of Node 1 as its sink node
Comparison of RPL and LIBP

Radio On Averages

LIBP uses less power amongst 10 Skymotes in relation to keeping their radios on, thus creating a more energy efficient network.


