Cooja: A Cross-Level Simulator for Contiki

Yang Deng
Background

- Simulator simplifies software development in controlled environment
  - Validate algorithm
  - Observe communication
  - Learn system behavior
- However, most of the simulators focus on specific level
  - Application Level: NS2/3, OMNet++
  - Operating System Level: TOSSIM
  - Machine Instruction Level: ATEMU
- Sometimes, hybrid simulation is needed. Thus, Cooja is introduced to bridge this gap.
Cooja Overview

- COOJA ensures that each simulated node only handles one event.
- In simulation loop, COOJA “ticks” every available node.
- Interfaces are hardware peripherals and enable COOJA to detect and trigger events; observers-subscribers model is employed.
- During a node tick, both before and after Contiki code actually executes, interfaces are allowed to check for new events.
- At the end of simulation loop the overall time is increased; stop or pause is allowed at this time.
- Support different level of simulation, implemented by 3 kinds of nodes: pure-Java nodes, native nodes, emulated nodes.
- For emulated nodes, each one is allowed to run for maximum a fixed period of time; then events are transferred via programming interface to and from COOJA.
Cooja Overview

Node 1
  ↓
Node 2
  ↓
Node 3
  ↓
Node X
  ↓
Node N
  ↓

Increase simulation time (stop/pause)

{ Pre-tick all sim interfaces
  Set memory
  Tick
  Get memory
  Post-tick all sim interfaces

Node Type

Pre-tick all sim interfaces
Set memory
Tick
Get memory
Post-tick all sim interfaces

Call `process_run()`

Direct
JNI
API
MSPSim Emulator

Java Function

Application Level
Pure-Java Nodes
Operating System Level
Native Nodes
Machine Instruction Level
Emulated Nodes
Operating System Level

- In native nodes node type acts as a link between a node and loaded Contiki
- COOJA makes JNI calls from Java Runtime to C Runtime
- JNI calls contain a few functions: initialize stack, copy and replace memory, tick the node and get address of variables
- Every node stores a copy of data memory (BSS and DATA segment) but all nodes of the same type share the same program code (TEXT segment)
- Data memory is copied to C Runtime before Contiki code executes and after that it is fetched back to Java Runtime
- Map-file is used to parse the address of variable and function in order to communicate with C Runtime; also used for debugging
- Stack is not copied as correct Contiki code should not use stack as a storage point between events (due to protothread)
Operating System Level

Loaded Contiki

JNI

Memory

Stack

BSS

DATA

TEXT

get/set memory

initialize

tick

Simulated Node B1

BSS

DATA

Memory Copy

Interfaces

Simulated Node Bn

BSS

DATA

Memory Copy

Interfaces

Simulated Node A1

BSS

DATA

Memory Copy

Interfaces

Simulated Node An

BSS

DATA

Memory Copy

Interfaces

Cooja

Node Type B

Node Type A

Java Runtime

C Runtime

get/set memory

get/set memory

get/set memory

get/set memory

initialize

initialize

tick

tick
Machine Instruction Level

- Emulated Nodes
  - MSPSim: Java-based instruction level emulator
  - External hardware is also emulated (TR1001, CC2420)
  - Command Line Interface and integration programming interface
Working Together

- **Workflow:** Write Same Code, Run Different nodes
On average, Pure-Java runs 1.7 times faster than native nodes and 50 times faster than emulated nodes.

<table>
<thead>
<tr>
<th></th>
<th>Pure-Java Nodes</th>
<th>Native Nodes</th>
<th>Emulated Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>Application</td>
<td>OS</td>
<td>Instruction</td>
</tr>
<tr>
<td>Contiki</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Deployable</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Speed</td>
<td>Fastest</td>
<td>Fast</td>
<td>Slowest</td>
</tr>
<tr>
<td>Memory</td>
<td>Smallest</td>
<td>Small</td>
<td>Largest</td>
</tr>
<tr>
<td>Usage</td>
<td>Develop / Validate algorithms</td>
<td>Normal cases</td>
<td>Cycle-accurate emulation</td>
</tr>
<tr>
<td>Scalability</td>
<td>$&gt;10^5$</td>
<td>$&gt;3\times10^4$</td>
<td>$&lt;100$</td>
</tr>
</tbody>
</table>
References


