TinyOS 2.x

Hongwei Zhang

http://www.cs.wayne.edu/~hzhang

Adapted from the IPSN’09 tutorial by
Stephen Dawson-Haggerty, Omprakash Gnawali, David Gay, Philip Levis, Răzvan Musăloiu-E., Kevin Klues, and John Regehr
What?

• An operating system for low power, embedded, wireless devices
  – Wireless sensor networks (WSNs)
  – Sensor-actuator networks
  – Embedded robotics

• Open source, open developer community

• http://www.tinyos.net
Goals

• Give you a high-level understanding of TinyOS’s structure and ideas
• Explain how to build applications
• Survey important libraries
  – Focus on very recent additions
Outline

- Basics
- TOSSIM
- Safe TinyOS
- Threads
- Protocols
Basics

Philip Levis (Stanford)
David Gay (Intel Research)
Outline

• *Components and interfaces*
  – Basic example

• Tasks
  – More complex example

• Compiling and toolchain
Outline

• *Components and interfaces*
  – Basic example

• Tasks
  – More complex example

• Compiling and toolchain
TinyOS Components

- TinyOS and its applications are in nesC
  - C dialect with extra features
- Basic unit of nesC code is a component
- Components connect via interfaces
  - Connections called “wiring”
Components

• A component is a file
  – names must match

• Modules are components that have variables and executable code

• Configurations are components that wire other components together
Component Example

- BlinkC wires BlinkP.Timer to TimerC.Timer

module BlinkP { ... }
implementation {
    int c;
    void increment() {c++;}
}

configuration BlinkC { ... }
implementation {
    components new TimerC();
    components BlinkP;
    BlinkP.Timer -> TimerC;
}
Singletons and Generics

- Singleton components are unique: they exist in a global namespace
- Generics are instantiated: each instantiation is a new, independent copy

```java
configuration BlinkC { ... }
implementation {
    components new TimerC();
    components BlinkP;
    BlinkP.Timer -> TimerC;
}
```
Interfaces

- Collections of related functions
- Define how components connect
- Interfaces are bi-directional: for A->B
  - Commands are from A to B
  - Events are from B to A
- Can have parameters (types)

```cpp
interface Timer<tag> {
    command void startOneShot(uint32_t period);
    command void startPeriodic(uint32_t period);
    event void fired();
}
```
Outline

• Components and interfaces
  – Basic example

• Tasks
  – More complex example

• Compiling and toolchain
Basic Example

• Goal: write an anti-theft device. Let’s start simple.

• Two parts:
  – Detecting theft.
    • Assume: thieves put the motes in their pockets.
    • So, a “dark” mote is a stolen mote.
    • Every N ms check if light sensor is below some threshold
  – Reporting theft.
    • Assume: bright flashing lights deter thieves.
    • Theft reporting algorithm: light the red LED for a little while!

• What we’ll see
  – Basic components, interfaces, wiring
  – Essential system interfaces for startup, timing, sensor sampling
module AntiTheftC {
uses interface Boot;
uses interface Timer<TMilli> as Check;
uses interface Read<uint16_t>;
}

implementation {

}
module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}
implementation {
    event void Boot.booted() {
        call Check.startPeriodic(1000);
    }
    event void Check.fired() {
        call Read.read();
    }
    event void Read.readDone(error_t ok, uint16_t val) {
        if (ok == SUCCESS && val < 200)
            theftLed();
    }
}

In TinyOS, all long-running operations are split-phase:
• A command starts the op: read
• An event signals op completion: readDone

interface Read<val_t> {
    command error_t read();
    event void readDone(error_t ok, val_t val);
}
module AntiTheftC {
    uses interface Boot;
    uses interface Timer<TMilli> as Check;
    uses interface Read<uint16_t>;
}

implementation {
    event void Boot.booted() {
        call Check.startPeriodic(1000);
    }
    event void Check.fired() {
        call Read.read();
    }
    event void Read.readDone(error_t ok, uint16_t val) {
        if (ok == SUCCESS && val < 200)
            theftLed();
    }
}

In TinyOS, all long-running operations are split-phase:
- A command starts the op: read
- An event signals op completion: readDone
Errors are signalled using the error_t type, typically
- Commands only allow one outstanding request
- Events report any problems occurring in the op
A configuration is a component built out of other components. It wires "used" to "provided" interfaces. It can instantiate generic components. It can itself provide and use interfaces.
Components

Diagram:

- AntiTheftC
  - Boot
  - Leds
  - Timer<TMilli>
  - Read<uint16_t>

  - MainC
  - LedsC
  - TimerMilliC (MyTimer)
  - PhotoC
Outline

• Components and interfaces
  – Basic example
• *Tasks and concurrency*
  – More complex example
• Compiling and toolchain
Tasks

- TinyOS has a single stack: long-running computation can reduce responsiveness
- Tasks: mechanism to defer computation
  - Tells TinyOS “do this later”
- Tasks run to completion
  - TinyOS scheduler runs them one by one in the order they post
  - Keep them short!
- Interrupts run on stack, can post tasks
Outline

- Components and interfaces
  - Basic example
- Tasks and concurrency
  - More complex example
- Compiling and toolchain
More Complex Application

• Let’s improve our anti-theft device. A clever thief could still steal our motes by keeping a light shining on them!
  – But the thief still needs to pick up a mote to steal it.
  – Theft Detection Algorithm 2: Every N ms, sample acceleration at 100Hz and check if variance above some threshold

• What we’ll see
  – (Relatively) high frequency sampling support
  – Use of tasks to defer computation-intensive activities
  – TinyOS execution model
Advanced Sensing, Tasks

uses interface ReadStream;
uint16_t accelSamples[ACCEL_SAMPLES];
event void Timer.fired() {
    call ReadStream.postBuffer(accelSamples, ACCEL_SAMPLES);
    call ReadStream.read(10000);
}

event void ReadStream.readDone(error_t ok, uint32_t actualPeriod) {
    if (ok == SUCCESS)
        post checkAcceleration();
}

task void checkAcceleration() {
    // check acceleration and report theft...
}

ReadStream is an interface for periodic sampling of a sensor into one or more buffers.
- postBuffer adds one or more buffers for sampling
- read starts the sampling operation
- readDone is signalled when the last buffer is full

interface ReadStream<val_t> {
    command error_t postBuffer(val_t* buf, uint16_t count);
    command error_t read(uint32_t period);
    event void readDone(error_t ok, uint32_t actualPeriod);
}
In readDone, we need to compute the variance of the sample. We defer this “computationally-intensive” operation to a separate task, using post. We then compute the variance and report theft.
TinyOS Execution Model

Stack

RealMainP
AntiTheftC
Timer
Alarm

Task Queue

SchedulerP

Timer
Alarm

Interrupt table

serial receive | H/W timer | A/D conv.
--- | --- | ---

Networking – “External” Types

```c
#include "antitheft.h"
module AntiTheftC {
    uses interface DisseminationValue<settings_t> as SettingsValue;

    #ifndef ANTITHEFT_H
    #define ANTITHEFT_H
typedef nx_struct {
        nx_uint8_t alert, detect;
        nx_uint16_t checkInterval;
    } settings_t;
    #endif

    settings_t settings;

    event void SettingsValue.changed() {
        const settings_t *newSettings = call SettingsValue.get();
        settings.detect = newSettings->detect;
        settings.alert = newSettings->alert;
        call Check.startPeriod(newSettings->checkInterval);
    }

    event void Timer.fired() {
        if (settings.detect & DETECT_DARK)
            call Read.read();
        if (settings.detect & DETECT_ACCEL)
            call ReadStream.postBuffer(accelSamples, ACCEL_SAMPLES);
            call ReadStream.read(10000);
    }
```
TinyOS/nesC Summary

- **Components and Interfaces**
  - Programs built by writing and wiring components
    - *modules* are components implemented in C
    - *configurations* are components written by assembling other components

- **Execution model**
  - Execution happens in a series of tasks (atomic with respect to each other) and interrupt handlers
  - No threads

- **System services: startup, timing, sensing (so far)**
  - (Mostly) represented by instantiatable generic components
    - This instantiation happens at compile-time! (think C++ templates)
  - All slow system requests are split-phase
Outline

• Components and interfaces
  – Basic example

• Tasks
  – More complex example

• Compiling and toolchain
The Toolchain

TinyOS

App

Native binary: 03 2F 77 9A F2 FF ...

PC Applications
The Toolchain

TinyOS

App

Native binary: 03 2F 77 9A F2 FF ...

PC Applications

Compile TinyOS applications
The Toolchain

TinyOS

App

Native binary: 03 2F 77 9A F2 FF ...

PC Applications

Install applications on motes
The Toolchain

TinyOS

Native binary: 03 2F 77 9A F2 FF ...

App

PC Applications

Build PC applications
The Toolchain

TinyOS

App

Native binary: 03 2F 77 9A F2 FF ...

Document TinyOS

PC Applications
The “Make” System

TinyOS

App

Native binary:
03 2F 77
9A F2 FF ...

make telosb install

automates nesC, C compilation, mote installation

PC Applications
“Make”: Compile Applications

int main() {
    scheduler_init();
    ...
}

Native binary:
03 2F 77 9A F2 FF ...

ncc

gcc
“Make”: Install Applications

Native binary:
03 2F 77
9A F2 FF
...

pybsl, uisp, etc

deluge
Build PC Applications

TinyOS

Packet formats, constants, etc

Native binary:
03 2F 77 9A F2 FF ...

Java, C, Python apps

Talk with motes
PC Applications: Extracting Information from TinyOS

TinyOS → mig
  ↓
  ↓
packets formats
  ↓
ncg
  ↓
  ↓
constants
ncd-dump
  ↓
  ↓
<the kitchen sink>
  ↓
Java, C or Python app
PC Applications: Talking to Motes

Java, C or Python app

packet libs

packet libs

sf
Document TinyOS

Component: tos.system.LedsC

configuration LedsC

The basic TinyOS LEDs abstraction.

Author:

Phil Buonadonna
David Gay
Philip Levis
Joe Polastre

Provides

interface Leds

Wiring

Leds

GeneralIO

generateGeneralIO

PlatformLedsC
TOSSIM

Răzvan Musăloiu-E. (JHU)
<table>
<thead>
<tr>
<th>What is TOSSIM?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete event simulator</td>
</tr>
<tr>
<td>ns2</td>
</tr>
</tbody>
</table>
Cycle-accurate simulators

Avrora, MSPSim
Two directions

Port
make PC a supported platform

Virtualize
simulate one of the supported platforms
### Features

- Simulates a MicaZ mote
  - ATmega128L (128KB ROM, 4KB RAM)
  - CC2420
- Uses CPM to model the radio noise
- Supports two programming interfaces:
  - Python
  - C++
Anatomy

<table>
<thead>
<tr>
<th>TOSSIM</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>tos/lib/tossim</td>
<td>Makefile</td>
</tr>
<tr>
<td>tos/chips/atm128/sim</td>
<td>*.nc</td>
</tr>
<tr>
<td>tos/chips/atm128/pins/sim</td>
<td>*.h</td>
</tr>
<tr>
<td>tos/chips/atm128/timer/sim</td>
<td></td>
</tr>
<tr>
<td>tos/chips/atm128/spi/sim</td>
<td></td>
</tr>
<tr>
<td>tos/platforms/mica/sim</td>
<td></td>
</tr>
<tr>
<td>tos/platforms/micaz/sim</td>
<td></td>
</tr>
<tr>
<td>tos/platforms/micaz/chips/cc2420/sim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simulation Driver</td>
</tr>
<tr>
<td></td>
<td>*.py</td>
</tr>
</tbody>
</table>
Quick Overview

Application

NesC

Glue

Simulation

Python

C++
The Building Process

$ make micaz sim

1. Generate an XML schema
2. Compile the application
3. Compile the Python support
4. Build a share object
5. Copying the Python support

$ ./sim.py

app.xml
sim.o
pytossim.o
tossim.o
c-support.o
_TOSSIMmodule.o
TOSSIM.py
TOSSIM.py

Tossim
Radio
Mote
Packet
Mac
<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNode()</td>
<td>TOSSIM.Mote</td>
</tr>
<tr>
<td>radio()</td>
<td>TOSSIM.Radio</td>
</tr>
<tr>
<td>newPacket()</td>
<td>TOSSIM.Packet</td>
</tr>
<tr>
<td>mac()</td>
<td>TOSSIM.Mac</td>
</tr>
<tr>
<td>runNextEvent()</td>
<td></td>
</tr>
<tr>
<td>ticksPerSecond()</td>
<td></td>
</tr>
<tr>
<td>time()</td>
<td></td>
</tr>
</tbody>
</table>
Simulating 10 seconds

```python
from TOSSIM import *

t = Tossim([])

...

while t.time() < 10*t.ticksPerSecond():
    t.runNextEvent()
```
**Syntax**

```
dbg(tag, format, arg1, arg2, ...);
```

**Example**

```
dbg("Trickle", "Starting time with time %u\n", timerVal);
```

**Python**

```
t = Tossim([])
t.addChannel("Trickle", sys.stdout)
```
Useful nesC Functions for TOSSIM simulation

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>sim_time_string()</td>
</tr>
<tr>
<td>sim_time_t</td>
<td>sim_time()</td>
</tr>
<tr>
<td>int</td>
<td>sim_random()</td>
</tr>
<tr>
<td>sim_time_t</td>
<td>sim_ticks_per_sec()</td>
</tr>
</tbody>
</table>

```plaintext
typedef long long int sim_time_t;
```
Closest-fit Pattern Matching (CPM)

Improving Wireless Simulation Through Noise Modeling
HyungJune Lee, Alberto Cerpa, and Philip Levis
IPSN 2007
TOSSIM.Radio

.add(source, destination, gain)

.connected(source, destination) → True/False

gain(source, destination)
TOSSIM.Mote

.bootAtTime(time)
.addNoiseTraceReading(noise)
.createNoiseModel()

.isOn() → True/False
.turnOn()/ .turnOff()
```python
from TOSSIM import *
t = Tossim([])
r = t.Radio()
mote0 = t.getNode(0)
mote1 = t.getNode(1)
mote2 = t.getNode(2)
r.add(0, 1, -10)
r.add(1, 0, -10)
r.add(1, 2, -50)
r.add(2, 1, -50)
```
Example (cont)

```python
noise = file("meyer-short.txt")
lines = noise.readlines()
for line in lines:
    str = line.strip()
    if (str != ":"):
        val = int(str)
        for m in [mote0, mote1, mote2]:
            m.addNoiseTraceReading(val)
for m in [mote0, mote1, mote2]:
    m.createNoiseModel()
```
### Other Features

- Injecting packets
- Inspecting internal variables
- C++ interface
- Debuging using gdb
Improvements

• **TossimLive**
  – SerialActiveMessageC

• **CC2420sim**
  – Multiple channels
  – PacketLink
  – CC2420Packet: .getRSSI(), .getLQI()
  – ReadRssi()
  – Flash support
Future

Parametrize the PRR/SNR curve based on packet size (in progress)

Support for multiple binary images (harder)
Safe TinyOS

John Regehr (Utah)
What is Safe TinyOS?

• Memory safe execution for TinyOS 2.1 apps
  – Compiler inserts safety checks
  – These checks trap pointer / array errors before they can corrupt memory

• Behavior of memory-safe applications is unchanged

• Why use Safe TinyOS?
  – Debugging pointer and array problems on motes can be extremely difficult
Using Safe TinyOS

- Must explicitly request safe compilation
  
  ```
  $ cd tinyos-2.x/apps/BaseStation
  $ make micaz safe
  ...
  18544 bytes in ROM
  1724 bytes in RAM
  $ make micaz
  ...
  14888 bytes in ROM
  1724 bytes in RAM
  ```
Designed to Fail

• In TinyOS 2.1:
  $ cd $TOSROOT/apps/tutorials/BlinkFail
  $ make micaz install

• The application dies after a few seconds
  – BlinkFailC.nc has an obvious memory bug

• Next try this:
  $ make micaz safe install

• After a few seconds the mote starts blinking its LEDs in funny patterns
• Default behavior on safety violation is to output a FLID (Fault Location IDentity) using the LEDs

• A FLID is 8 digits in base-4
  – No LEDs lit = 0
  – 1 LED lit = 1
  – 2 LEDs lit = 2
  – 3 LEDs lit = 3

• A tool decodes FLIDs into error messages
$ tos-decode-flid ./build/micaz/flids.txt 00001020

Deputy error message for flid 0x0048:

\[
\begin{align*}
\text{BlinkFailC\_a} & \leq \text{BlinkFailC\_a} + \text{BlinkFailC\_i++} + 1 \\
\text{(with no overflow): BlinkFailC.nc:70:}
\end{align*}
\]

Assertion failed in CPtrArithAccess: \[
\text{BlinkFailC\_a} + \text{BlinkFailC\_i++} + 1 \leq \text{BlinkFailC\_a} + 10 \text{ (with no overflow)}
\]
Safe Components

• Safety is “opt in” at the level of nesC components

• This component is compiled as safe code:
  
  ```
  generic module Simple ArbiterP() @safe() { … }
  ```

• These components are “trusted” code:
  
  ```
  generic module Simple ArbiterP() @unsafe() { … }
  generic module Simple ArbiterP() { … }
  ```

• Trusted code is compiled w/o safety checks
Porting Code to Safe TinyOS

- **Recommended strategy**
  1. Annotate a component as `@safe()`
  2. Compile application in safe mode
  3. Fix warnings / errors
  4. Repeat until no trusted components remain

- **Arrays and pointers require annotations**
  - Annotations are for Deputy, the safe C compiler behind Safe TinyOS
  - Purpose of annotations is to link memory regions with their bounds information
Annotation 1

• To declare `msg`, which always refers to a valid `message_t`

```c
message_t* ONE msg = ...;
```

• Or if `msg` may be null

```c
message_t* ONE_NOK msg;
```

• Most annotations have a `_NOK` form
  – But avoid using it when possible
• To declare `uartQueue` as an array of 10 pointers to `message_t`
  – Where each element of the array must at all times refer to a valid `message_t`

```c
message_t* ONE uartQueue[10];
```
• To declare `reqBuf` as a pointer that always points to a valid block of at least `reqBytes`

```
uint8_t *COUNT(reqBytes) reqBuf;
```

• Array dereferencing / pointer arithmetic can be done on `reqBuf`:
  - `reqBuf[0]` is legal
  - `reqBuf[reqBytes-1]` is legal
  - `reqBuf[reqBytes]` results in a safety violation
• Multiple-indirect pointers require an annotation at each level:

\[
\text{int } \ast\text{ONE } \ast\text{ONE pp } = \ldots;
\]

• However, these are uncommon in TinyOS
Annotation 5

• Trusted cast
  – tells Deputy to just trust the programmer
  – is needed to perform casts that are safe, but are beyond the reach of Deputy's type system

```c
cc2420_header_t* ONE x = TCAST(
  cc2420_header_t* ONE,
  (uint8_t *)msg +
  offsetof(message_t, data) -
  sizeof(cc2420_header_t)
);
```
• The `getPayload()` command from the Packet interface might be annotated like this:

```c
command void* COUNT_NOK(len)
getPayload (message_t* ONE msg,
            uint8_t len);
```
### Interface Annotation 2

- However, `tinyos-2.x/tos/interfaces/Packet.nc` contains:
  
  ```c
  * @param 'message_t* ONE msg' ...
  * @param len ...
  * @return 'void* COUNT_NOK(len)' ...
  */
  
  command void* getPayload (message_t* msg, uint8_t len);
  ```

- nesC allows you to put annotations in documentation comments
Safe TinyOS Summary

- Safe execution is useful
- Safety annotations are good documentation
- Most Mica2, MicaZ, TelosB apps and core services are safe
- Safe TinyOS Tutorial:
  - [http://docs.tinyos.net/index.php/Safe_TinyOS](http://docs.tinyos.net/index.php/Safe_TinyOS)
Threads

Kevin Klues (UCB)
The Great Divide

- Event-Based Execution
  - More efficient
  - Less RAM usage
  - More complex

- Thread-Based Execution
  - Less Efficient
  - More RAM Usage
  - Less Complex

TOSTThreads aims to resolve this fundamental tension
TOSTThreads in a Nutshell

- Natural extension to the existing TinyOS concurrency model
- Implements Full-Fledged Threads Library
- Introduces Minimal Disruption to TinyOS
- Provides Flexible Event-based / Thread-based Code Boundary
- Enables Dynamic Linking and Loading of Application Binaries at Runtime
- Standard C and nesC based APIs
Architecture Overview

Application Threads

System Calls

Task Scheduler

TinyOS Thread

Thread Scheduler
configuration BlinkAppC {
}

implementation {
    components MainC, BlinkC, LedsC;
    components new ThreadC(STACK_SIZE);

    MainC.Boot <- BlinkC;
    BlinkC.Thread -> ThreadC;
    BlinkC.Leds -> LedsC;
}

module BlinkC {
    uses {
        interface Boot;
        interface Thread;
        interface Leds;
    }

    implementation {
        event void Boot.booted() {
            call Thread.start(NULL);
        }

        event void Thread.run(void* arg) {
            for(;;) {
                call Leds.led0Toggle();
                call Thread.sleep(BLINK_PERIOD);
            }
        }
    }
}
# Blink Example (standard C)

```c
#include "tosthread.h"
#include "tosthread_leds.h"

// Initialize variables associated with a thread
tosthread_t blink;
void blink_thread(void* arg);

void tosthread_main(void* arg) {
    tosthread_create(&blink, blink_thread, NULL, STACK_SIZE);
}
void blink_thread(void* arg) {
    for(;;) {
        led0Toggle();
        tosthread_sleep(BLINK_PERIOD);
    }
}
```
Modifications to TinyOS

- Change in boot sequence
- Small change in TinyOS task scheduler
- Additional post-amble in the interrupt sequence
**Boot Sequence**

**Standard TinyOS Boot**

```java
event void TinyOS.booted() {
    atomic {
        platform_bootstrap();

        call Scheduler.init;

        call PlatformInit.init();
        while (call Scheduler.runNextTask());

        call SoftwareInit.init();
        while (call Scheduler.runNextTask());
    }
    signal Boot.booted();

    /* Spin in the Scheduler */
    call Scheduler.taskLoop();
}
```

**Main**

```java
int main() {
    signal TinyOS.booted();

    //Should never get here
    return -1;
}
```
**Boot Sequence**

**Thread Scheduler Boot**

```cpp
event void ThreadScheduler.booted() {
    setup_TinyOS_in_kernel_thread();
    signal TinyOSBoot.booted();
}
```

**New Main**

```cpp
int main() {
    signal ThreadScheduler.booted();

    //Should never get here
    return -1;
}
```
Task Scheduler

**Original**

```c
command void Scheduler.taskLoop() {
    for (;;) {
        uint8_t nextTask;
        atomic {
            while ((nextTask = popTask()) == NO_TASK))
                call McuSleep.sleep();
        }
        signal TaskBasic.runTask[nextTask]();
    }
}
```

**New**

```c
command void Scheduler.taskLoop() {
    for (;;) {
        uint8_t nextTask;
        atomic {
            while ((nextTask = popTask()) == NO_TASK))
                call ThreadScheduler.suspendThread(TOS_THREAD_ID);
        }
        signal TaskBasic.runTask[nextTask]();
    }
}
```
void interruptCurrentThread() {
    if (call TaskScheduler.hasTasks() ) {
        call ThreadScheduler.wakeupThread(TOS_THREAD_ID);
        call ThreadScheduler.interruptCurrentThread();
    }
}

TOSH_SIGNAL(ADC_VECTOR) {
    signal SIGNAL_ADC_VECTOR.fired();
    atomic interruptCurrentThread();
}

TOSH_SIGNAL(DACDMA_VECTOR) {
    signal SIGNAL_DACDMA_VECTOR.fired();
    atomic interruptCurrentThread();
}

....
....

void interruptCurrentThread() {
    if (call TaskScheduler.hasTasks() ) {
        call ThreadScheduler.wakeupThread(TOS_THREAD_ID);
        call ThreadScheduler.interruptCurrentThread();
    }
}
System Calls

Application Thread System Calls

- Send
- Receive
- Sense
- Block Storage

Task Queue

- Timer
- Receive
- Routing
- Arbiter

TinyOS Thread

System Call Task
Resources

- TOSThreads Tutorial
  http://docs.tinyos.net/index.php/TOSThreads_Tutorial

- TOSThreads TEP

- Source Code
  System code: tinyos-2.x/tos/lib/tosthreads
  Example Applications: tinyos-2.x/apps/tosthreads
Protocols

Omprakash Gnawali (USC)
Protocols in TinyOS 2.1

- **Network Protocols**
  - Collection: CTP, MultihopLQI
  - Dissemination: Drip, DIP

- **Time Synchronization (FTSP)**

- **Over-the-air programming (Deluge)**
Collection

- Collect data from the network to one or a small number of roots
- One of many traffic classes
- Available: MultihopLQI and CTP
MultihopLQI

- Mostly tested and used on platforms with CC2420
  - MicaZ, TelosB, ...
- Small code footprint
- tos/lib/net/lqi
<table>
<thead>
<tr>
<th>CTP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform independent</strong></td>
</tr>
<tr>
<td><strong>More consistent performance than with MultihopLQI</strong></td>
</tr>
<tr>
<td><strong>Code footprint can be a concern</strong></td>
</tr>
<tr>
<td><strong>tos/lib/net/ctp</strong></td>
</tr>
</tbody>
</table>
CTP Link Estimator

- Platform independent
  - Beacons and data packets
- Bi-directional ETX estimate
- Does not originate beacons itself
- Accurate but also agile
CTP Router

- ETX path metric
- Beacon interval can be 64 ms-x mins
- Select new path if better by at least 1.5 ETX
- Alternate parents
<table>
<thead>
<tr>
<th>CTP Forwarder</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Duplicate suppression</td>
</tr>
<tr>
<td>• Retransmissions</td>
</tr>
<tr>
<td>• Loops trigger route updates</td>
</tr>
<tr>
<td>• Forward through alternate parents</td>
</tr>
</tbody>
</table>
CTP Reliability
Dissemination

- Send data to all the nodes
  - Commands, configuration parameters
- Efficient and fast
- Available protocols – Drip and DIP
Drip

- Fast and efficient for small number of items
- Trickle timers for advertisements
- Suppression
- tos/lib/net/drip
DIP

- Efficiently Disseminates large number of items (can not fit in one packet)
- Use hashes and version vectors to detect and identify updates to the values
- tos/lib/net/dip
Deluge

- Over-the-air programming
- Disseminates code
- Programs the nodes
Deluge Details

- Supports Tmote Sky/EPIC and MicaZ.
- Bulk dissemination on top of Drip
- Python tools
- Support for MIB600. (new)
- tos/lib/net/Deluge, tos/lib/tosboot
Time Synchronization

• Global time on all the nodes
• Node with smallest id becomes the root
• Flooding Time Synchronization Protocol (FTSP)
• tos/lib/ftsp
Assignment

• Study TinyOS via the detailed tutorials at
  – http://docs.tinyos.net/tinywiki/index.php/TinyOS_Tutorials

• Information at the following site may well be helpful too
  – http://www.tinyos.net/tinyos-2.x/doc/
  – http://docs.tinyos.net/tinywiki/index.php/Main_Page
TinyOS Tutorials

These brief tutorials are intended to get you started with TinyOS. The later tutorials go a little deeper into some of the more advanced areas.

Contents [hide]

1 Working Group Tutorials
   1.1 Getting Started with TinyOS
   1.2 Modules and the TinyOS Execution Model
   1.3 Mote-mote radio communication
   1.4 Mote-PC serial communication and SerialForwarder
   1.5 Sensing
      1.5.1 ADC
   1.6 Boot Sequence
   1.7 Storage
   1.8 Resource Arbitration and Power Management
   1.9 Concurrency
   1.10 Platforms
   1.11 TOSSIM
   1.12 Network Protocols
   1.13 TinyOS Toolchain
   1.14 Building a simple but full-featured application
   1.15 The TinyOS printf Library
   1.16 Writing Low-Power Applications
   1.17 TOSThreads Tutorial
   1.18 CC2420 Security Tutorial
2 Other Tutorials
   2.1 lpsn2009-tutorial
3 User Contributed Tutorials
   3.1 Platform Creation and Testing
   3.2 Rssi Demo
Installing TinyOS 2.1.1

TinyOS has numerous improvements to TinyOS 2.1. Its features include:

- Support for the epic, mulle, and shimmer2 platforms,
- Support for 6lowpan, an IPv6 networking layer within the mote network,
- Support for simple, uniform low-power networking across many protocols,
- Improvements to many existing services and protocols, including the inclusion of a new disrd

More information can be found in the release notes.

Officially Supported Methods

- Full System:
  - One step installation with a Live CD, (doesn't currently work)
- Windows:
  - Manual installation using cygwin and RPM packages
  - Running a XubunTOS Virtual Machine Image in VMware Player
- Linux:
  - Manual installation using RPM packages
  - Automatic installation for debian systems using the TinyOS debian repository
  - Running a XubunTOS Virtual Machine Image in VMware Player

User Contributed Methods

- One step Install TinyOS 2 1.1 OSIAN IPv6: Ubuntu 10.04 Bootable DVD (32-bit)
Lab#0 (optional)

- Install TinyOS on your local machine, or use the TinyOS is CS labs
- Run the application **Blink** in TOSSIM
- Reference website: [http://www.tinyos.net/](http://www.tinyos.net/)