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

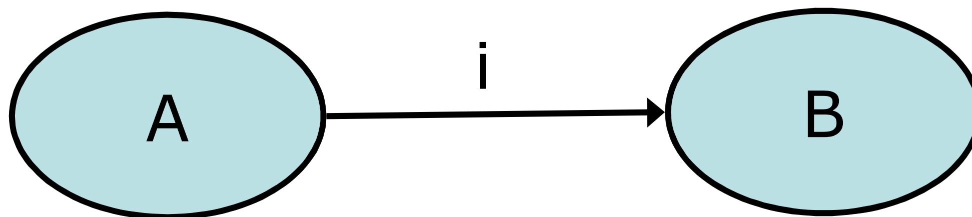
- *Components and interfaces*
  - Basic example
- Tasks
  - More complex example
- Compiling and toolchain

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”

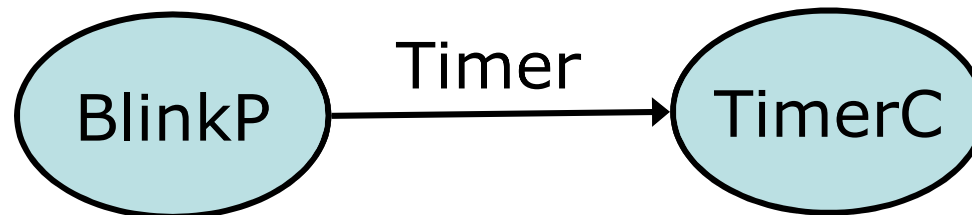


Components

- 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

```
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)

```
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

The Basics – Let's Get Started

```
module AntiTheftC {
  uses interface Boot;
  uses interface Timer<TMilli> as Check;
  uses interface Read<uint16_t>;
}
implementa
  event voi
    call Che
  }
  event voi
    call Rea
  }
  event void Read.readDone(error_
    if (ok == SUCCESS && val < 20
      theftLed();
  }
}
```

```
interface Boot {
  /* Signaled when OS booted */
  event void booted();
}
```

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

Components start with a *signature* specifying
- the interfaces *provided* by the component
- the interfaces *used* by the component

A module is a component implemented in C
- with functions implementing commands and events
- and extensions to call commands, events

The Basics – Split-Phase Ops

```
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);
}
```

The Basics – Split-Phase Ops

```
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

```
interface Read<val_t> {
  command error_t read();
  event void readDone(error_t ok, val_t val);
}
```

The Basics – Configurations

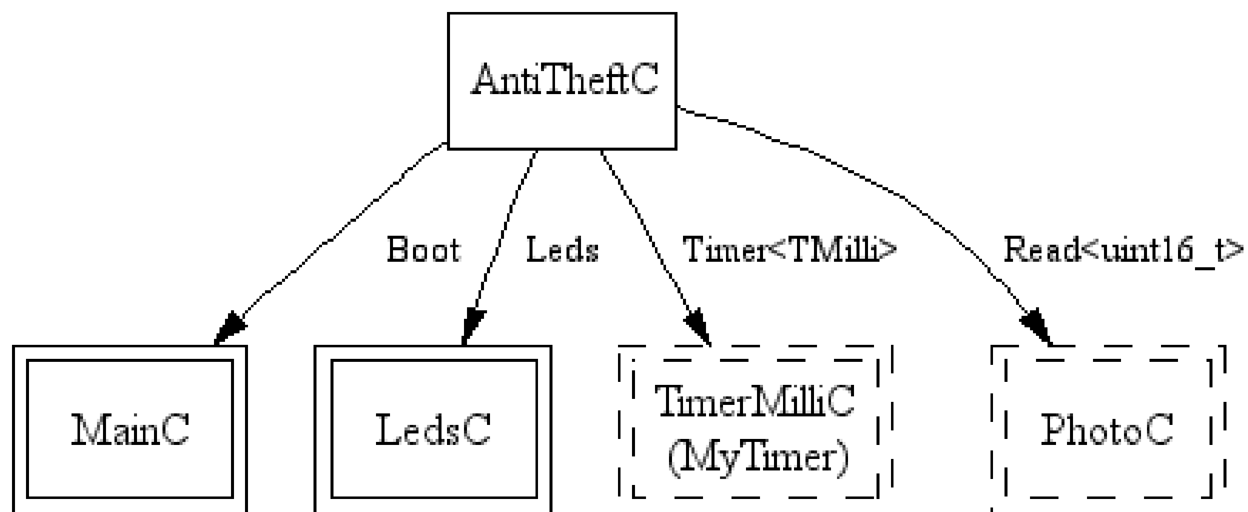
```
configuration AntiThe
implementation
{
  components AntiThe

  AntiTheftC.Boo
  AntiTheftC.Led

  components new TimerMilliC() as MyTimer;
  AntiTheftC.Check -> MyTimer;

  components  new PhotoC()
  AntiTheftC.Read -> PhotoC
}
```

```
generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
```

```
generic configuration PhotoC() {
  provides interface Read;
}
implementation { ... }
```

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



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.readD
  if (ok == SUCCESS)
    post checkAcceleration();
}

task void checkAcceleration() {
```

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

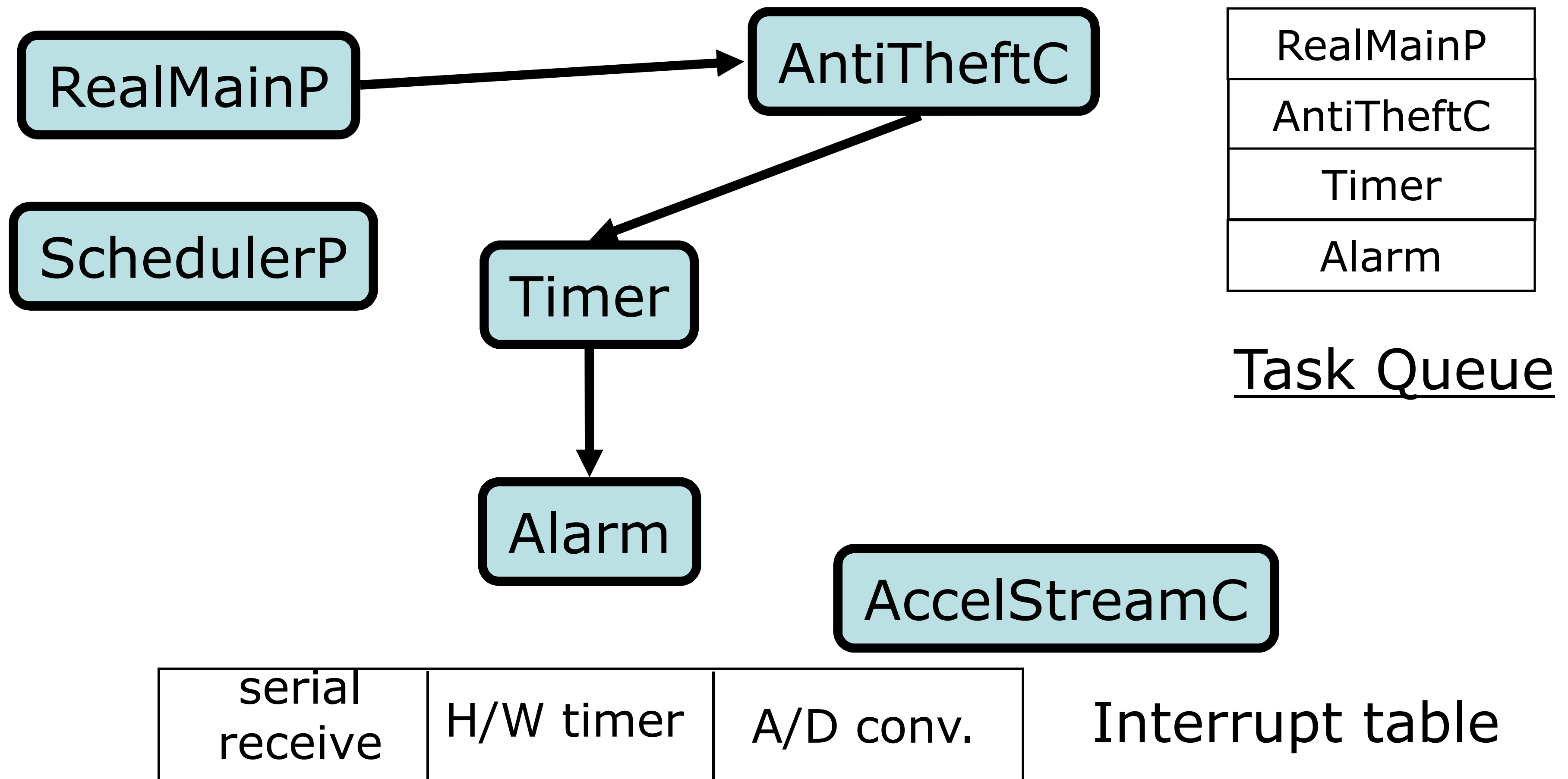
```
interface ReadStream<val_t> {
  command error_t postBuffer(val_t* buf, uint16_t count);
  command error_t read(uint32_t period);
  event void readDone(error_t ok, uint32_t actualPeriod);
}
```

Advanced Sensing, Tasks

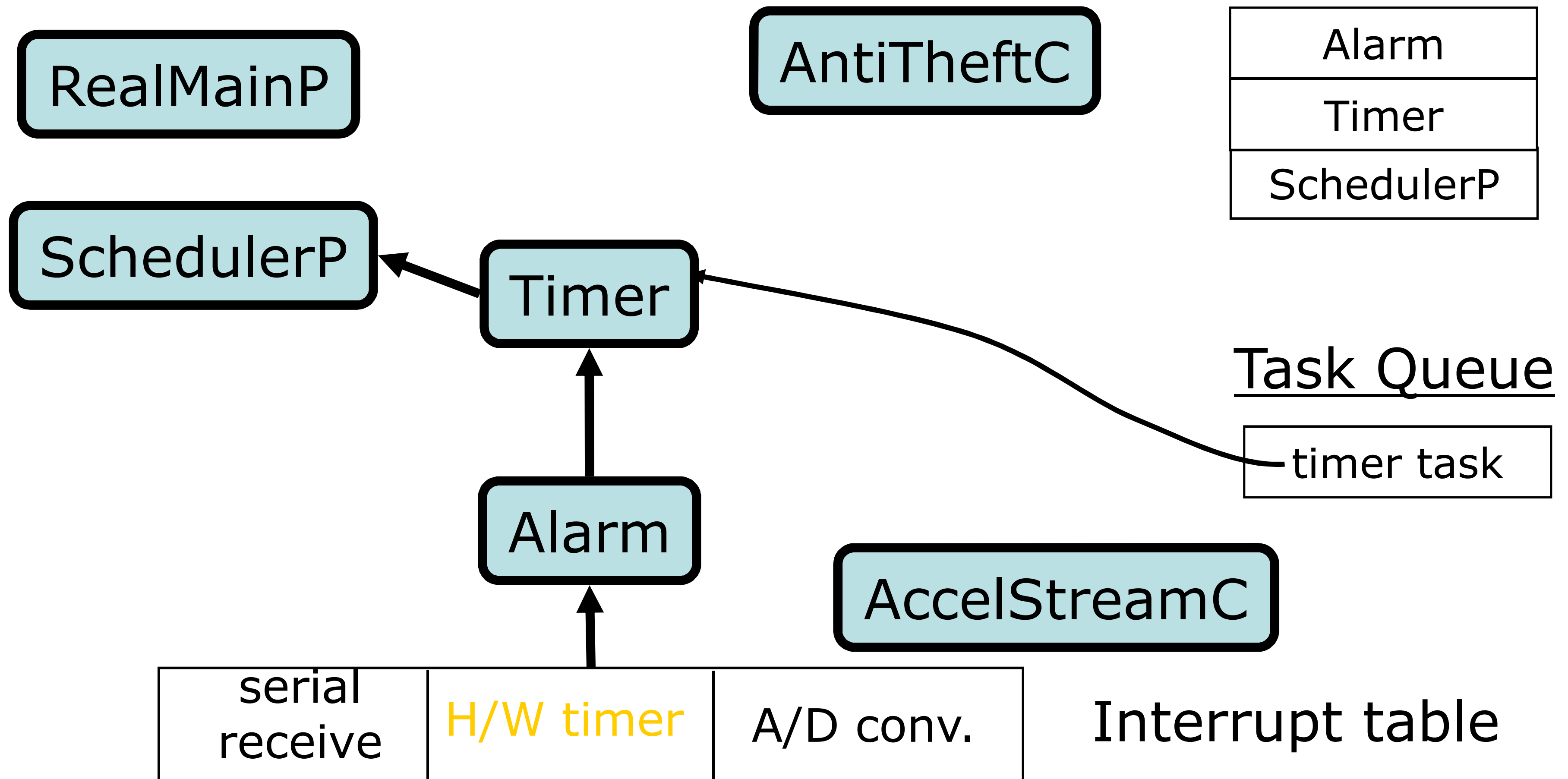
```
uint16_t accelSamples[SAMPLES];
event void ReadStream.readDone(error_t ok, uint32_t actualPeriod) {
  if (ok == SUCCESS)
    post checkAcceleration();
}
task void checkAcceleration() {
   uint16_t i, avg, var;

   for (avg = 0, i = 0; i < SAMPLES; i++)
     avg += accelSamples[i];
   avg /= SAMPLES;

   for (var = 0, i = 0; i < SAMPL
     {
       int16_t diff = accelSample
       var += diff * diff;
     }
   if (var > 4 * SAMPLES) theftLed();
}
```

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.

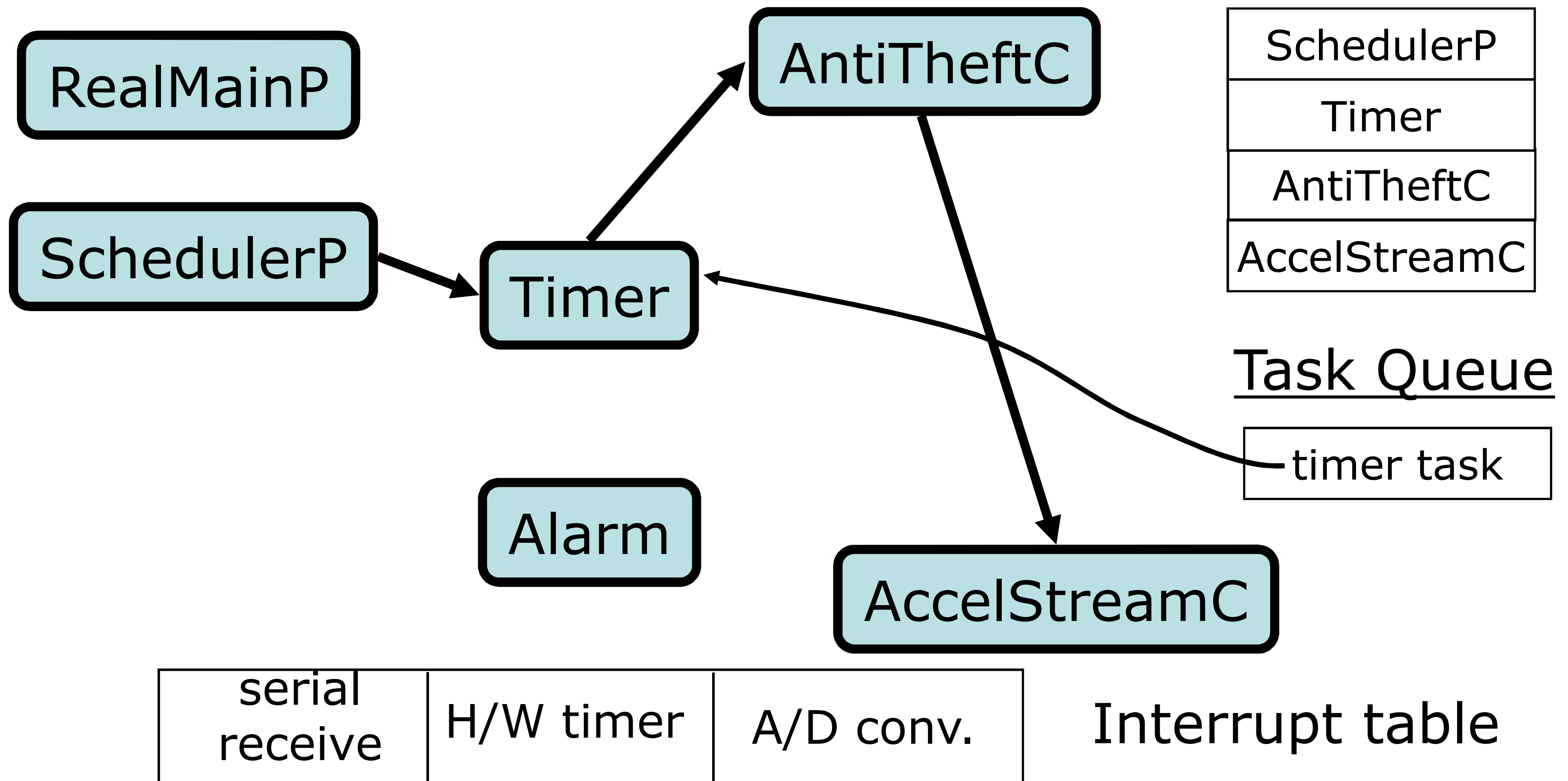
TinyOS Execution Model



TinyOS Execution Model



TinyOS Execution Model



Networking – “External” Types

```
#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
```

```
...gsValue.get();
...ckInterval);
  event void Timer.fired() {
    if (settings.detect & DETECT_...
      call Read.read();
    if (settings.detect & DETECT_...
        call ReadStream.postBuffer...
        call ReadStream.read(1000...
    }
}
```

External types (nx_...) provide C-like access, but:

- platform-independent layout and endianness gives interoperability
- no alignment restrictions means they can easily be used in network buffers
- compiled to individual byte read/writes

TinyOS/nesC Summary

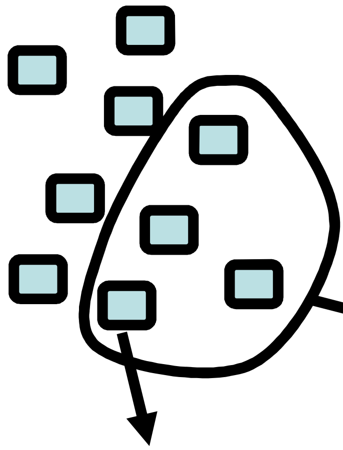
- Components and Interfaces
  - Programs built by writing and wiring components
    - *modules* are components implemented in C
    - *configurations* are components written by assembling other components
- Execution model
  - Execution happens in a series of tasks (atomic with respect to each other) 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

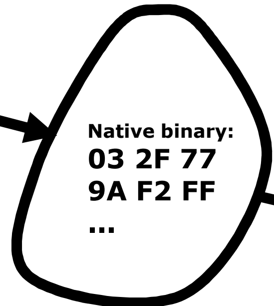
- Components and interfaces
  - Basic example
- Tasks
  - More complex example
- Compiling and toolchain

The Toolchain

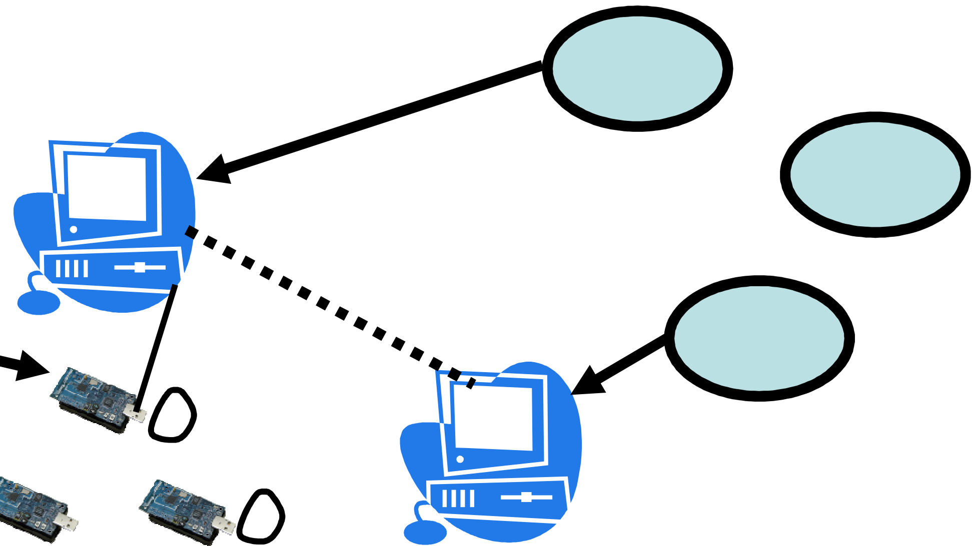
TinyOS

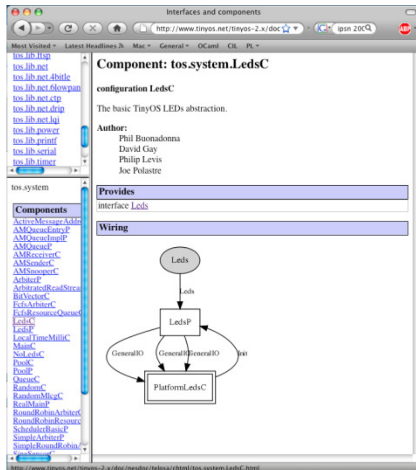


App

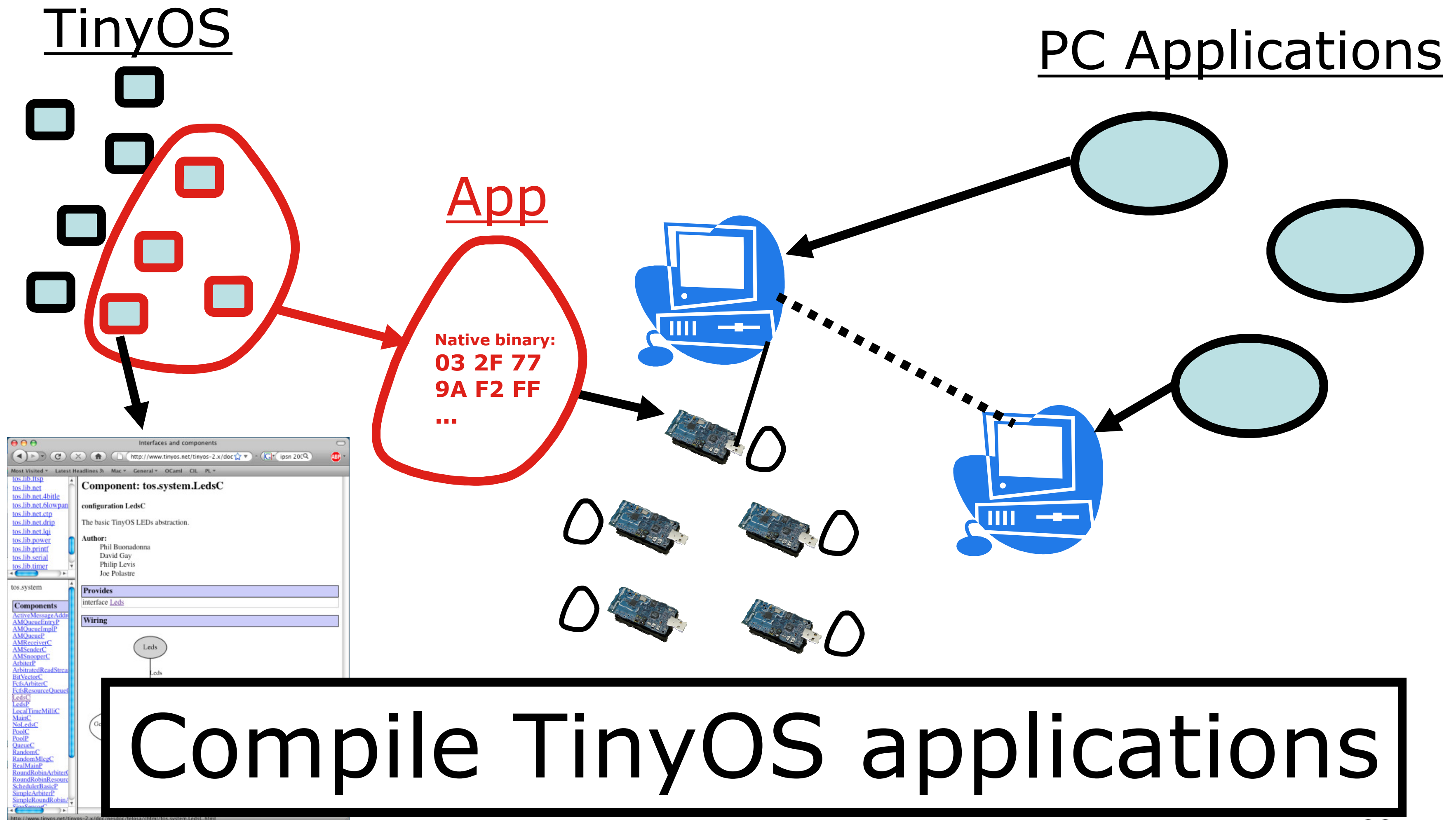


PC Applications

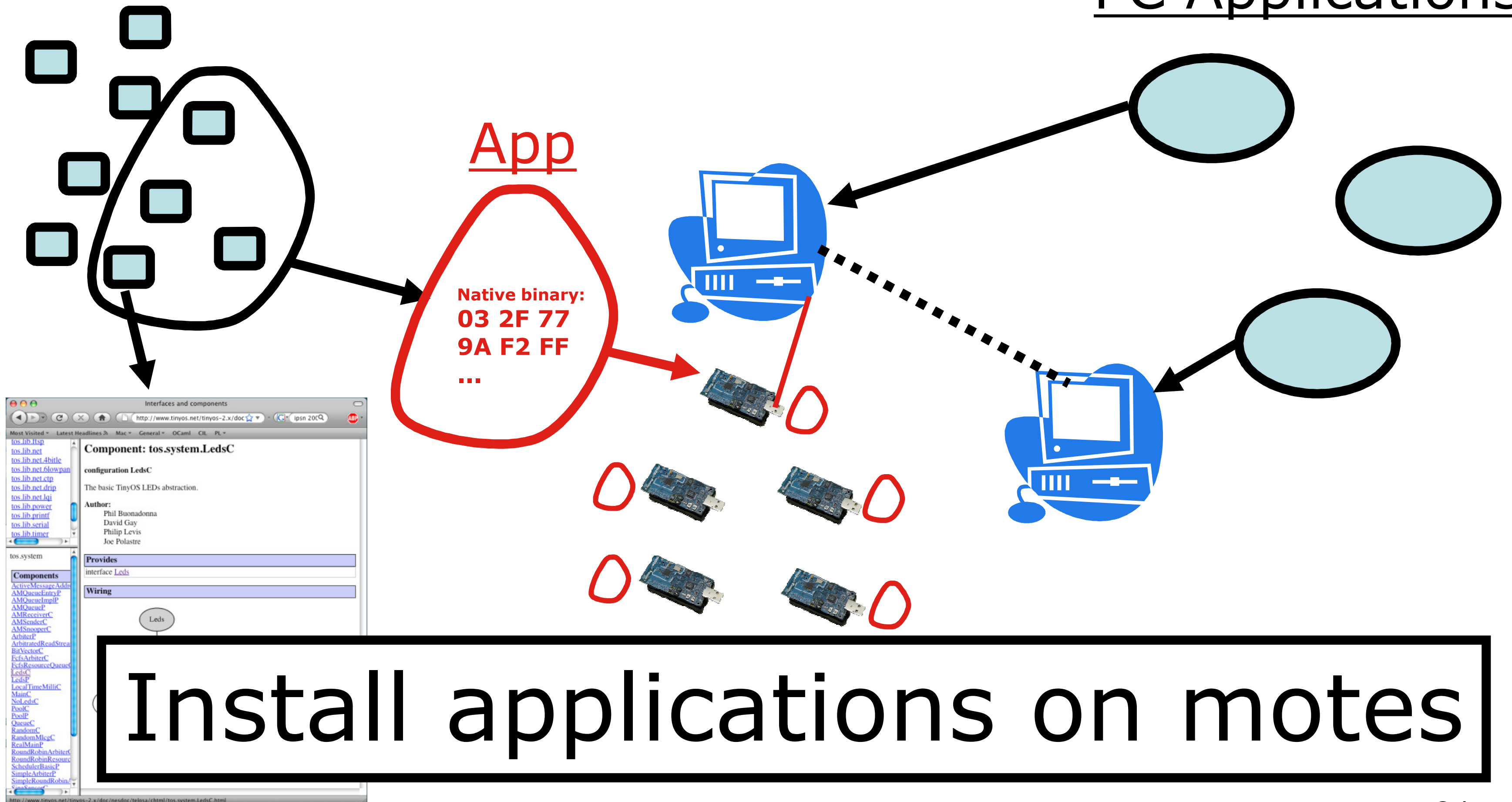




The Toolchain

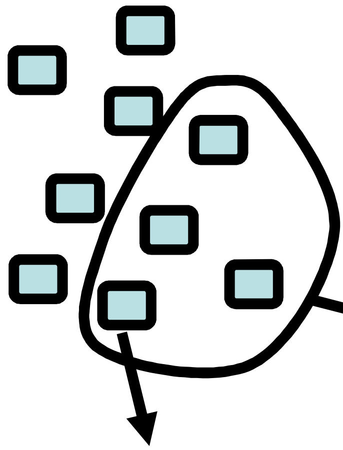


The Toolchain

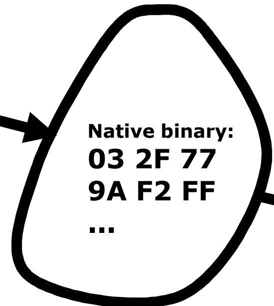


The Toolchain

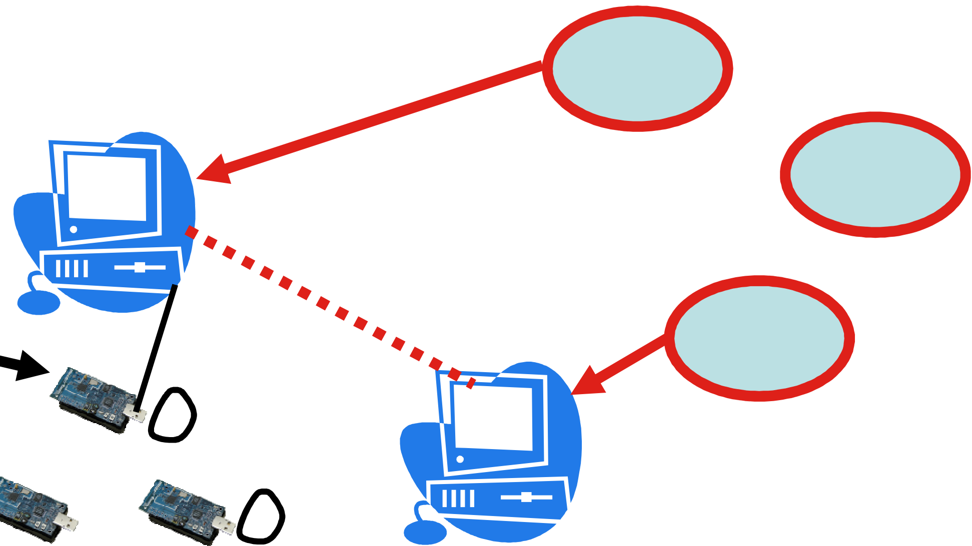
TinyOS

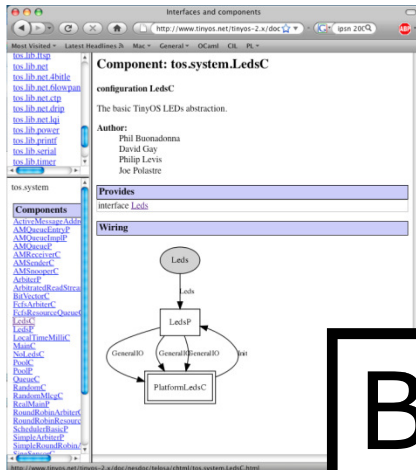


App



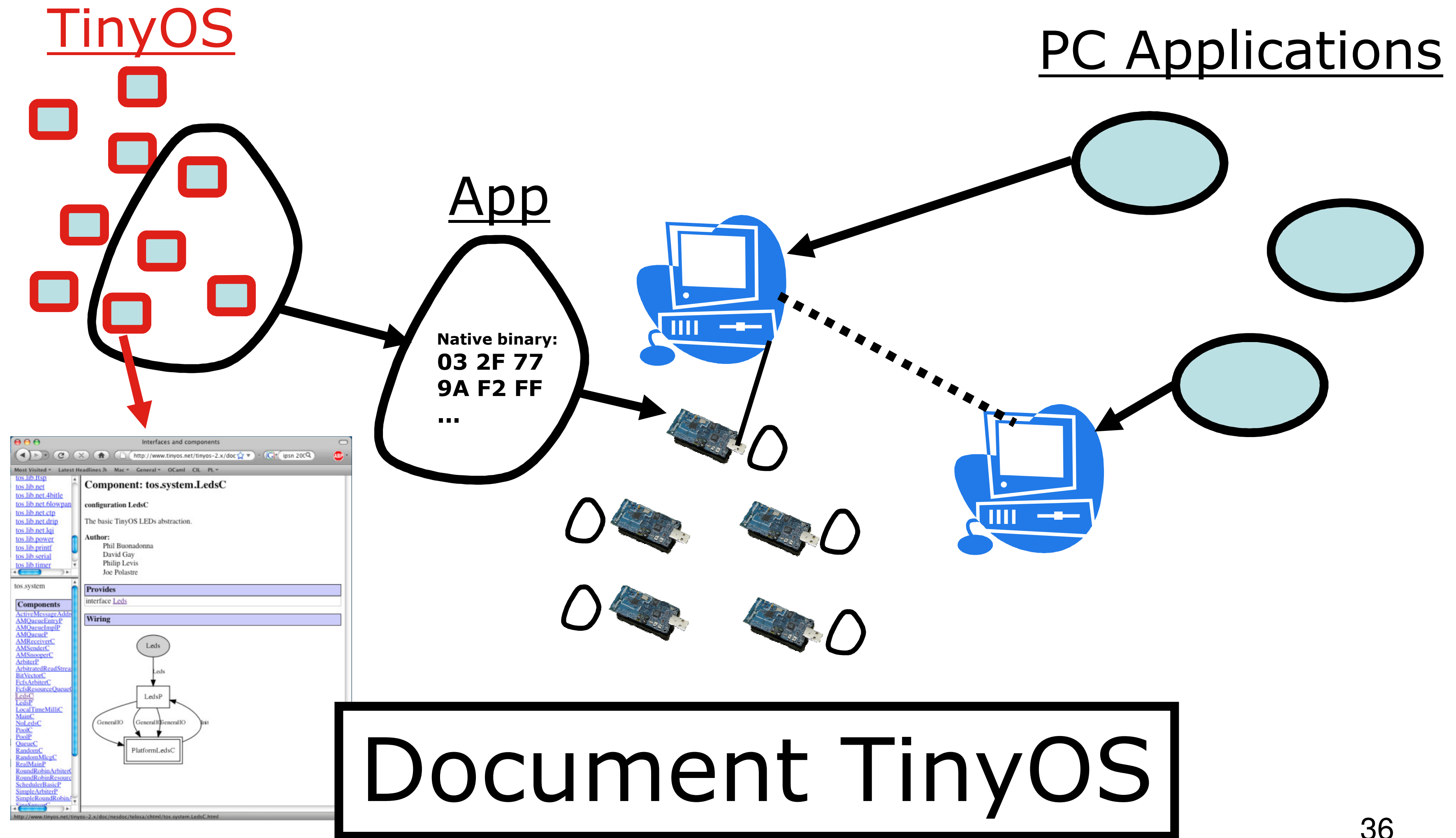
PC Applications





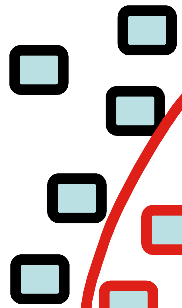
Build PC applications

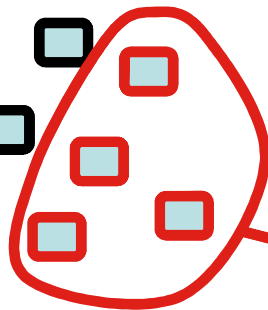
The Toolchain



The "Make" System

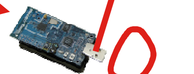
TinyOS

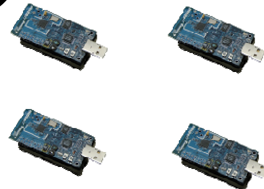


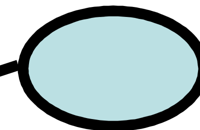


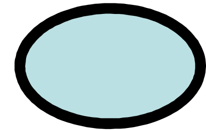
App

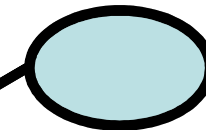
PC Applications







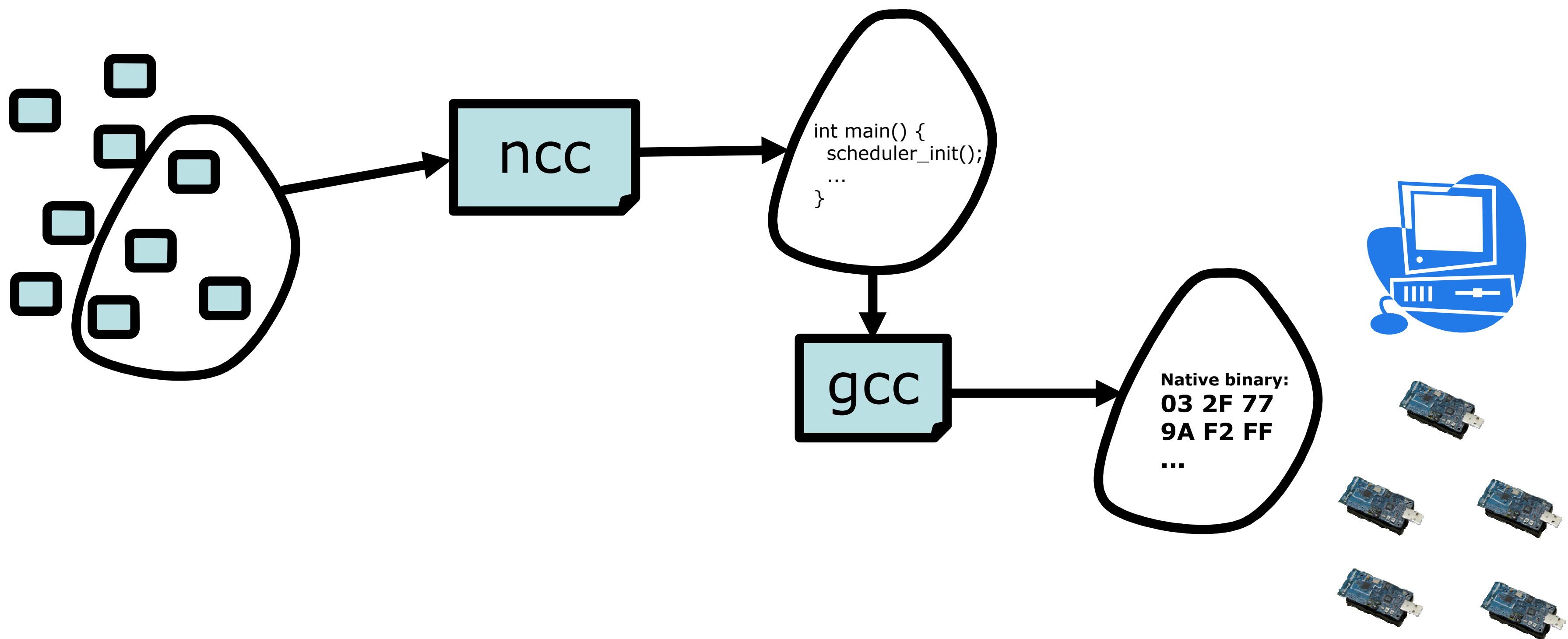




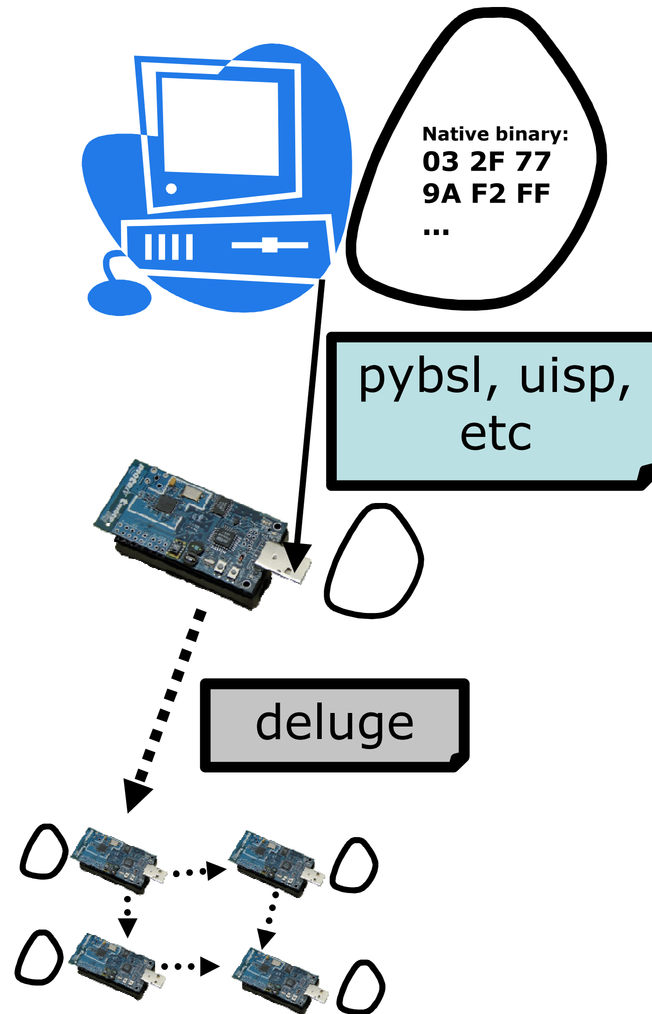
make telosb install

automates nesC, C compilation,
mote installation

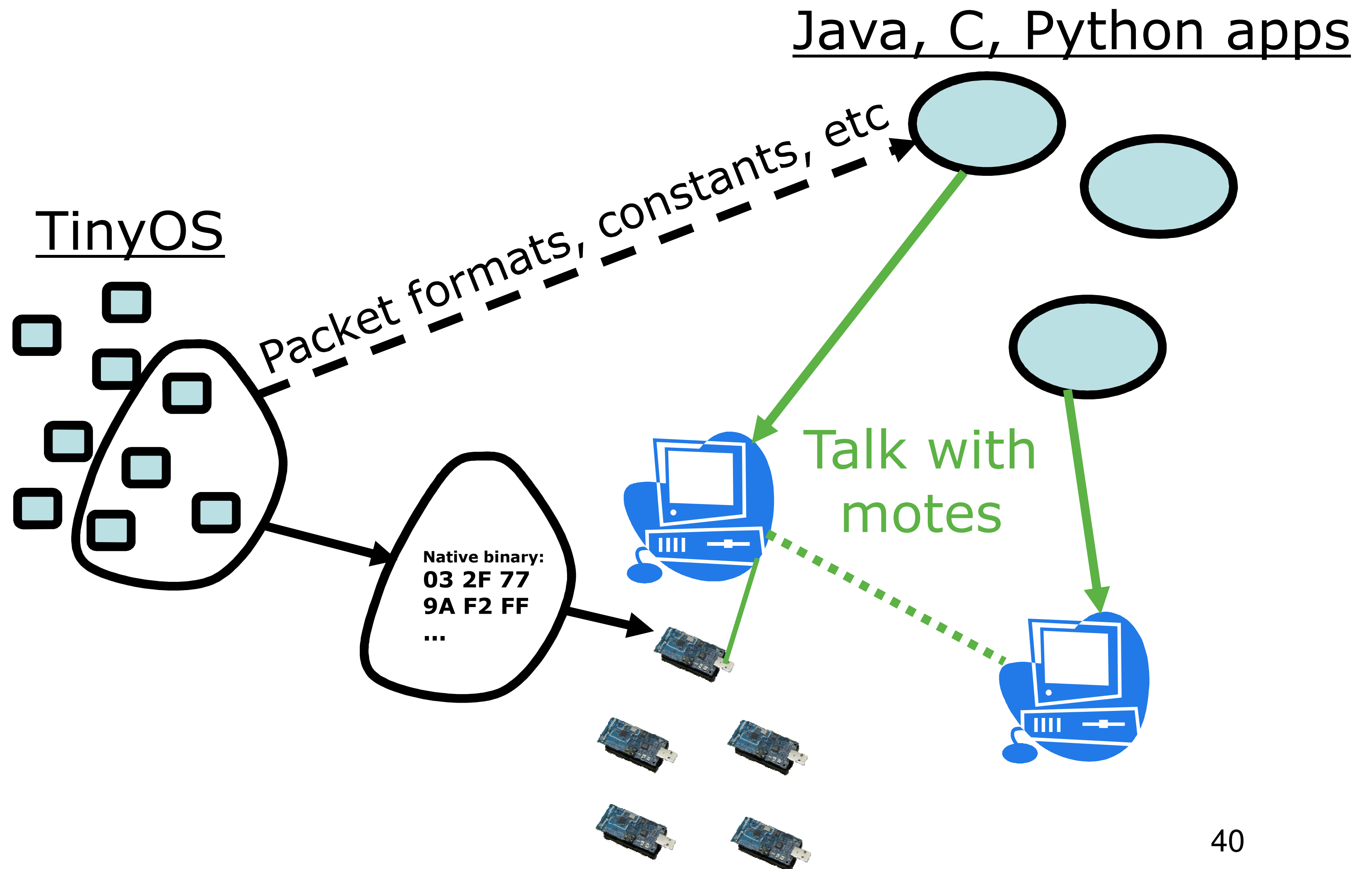
"Make": Compile Applications



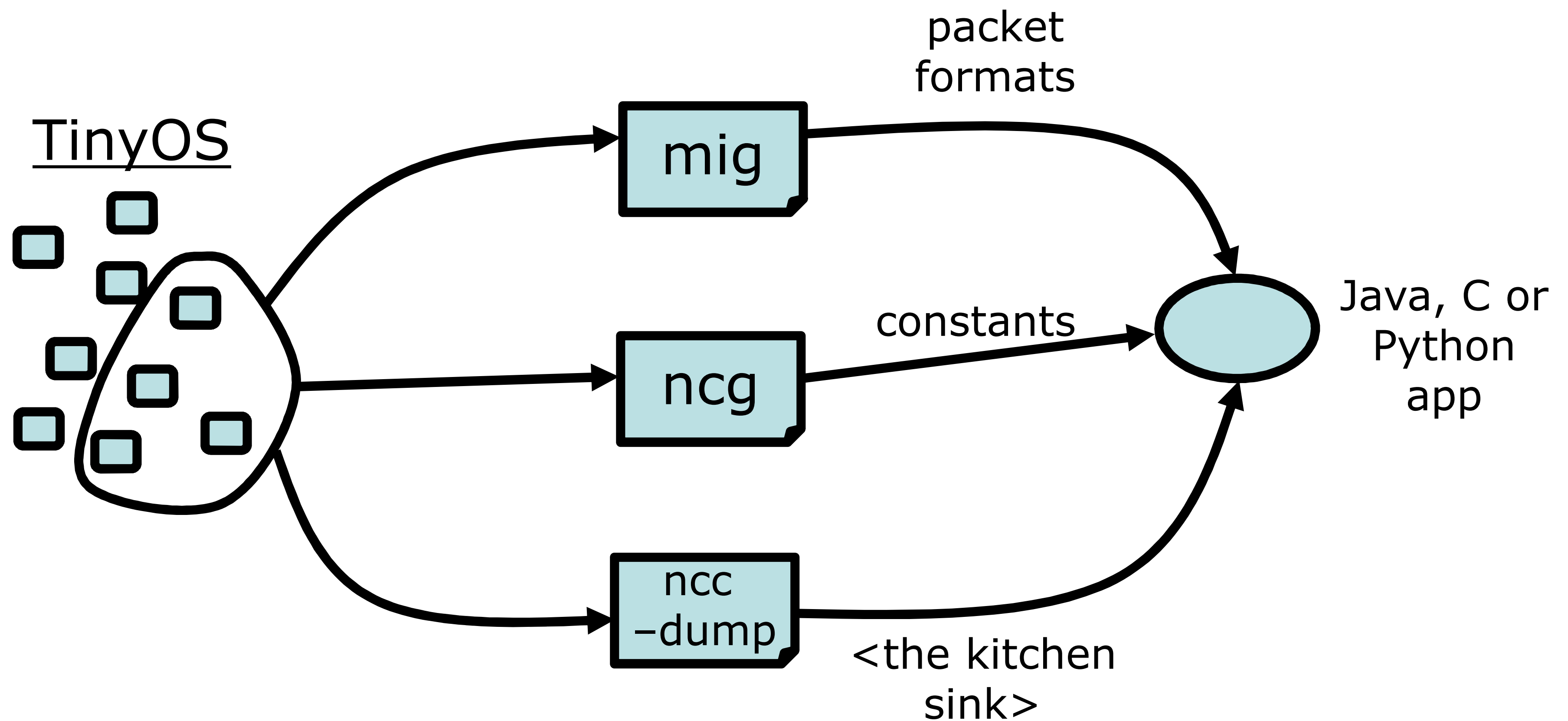
“Make”: Install Applications



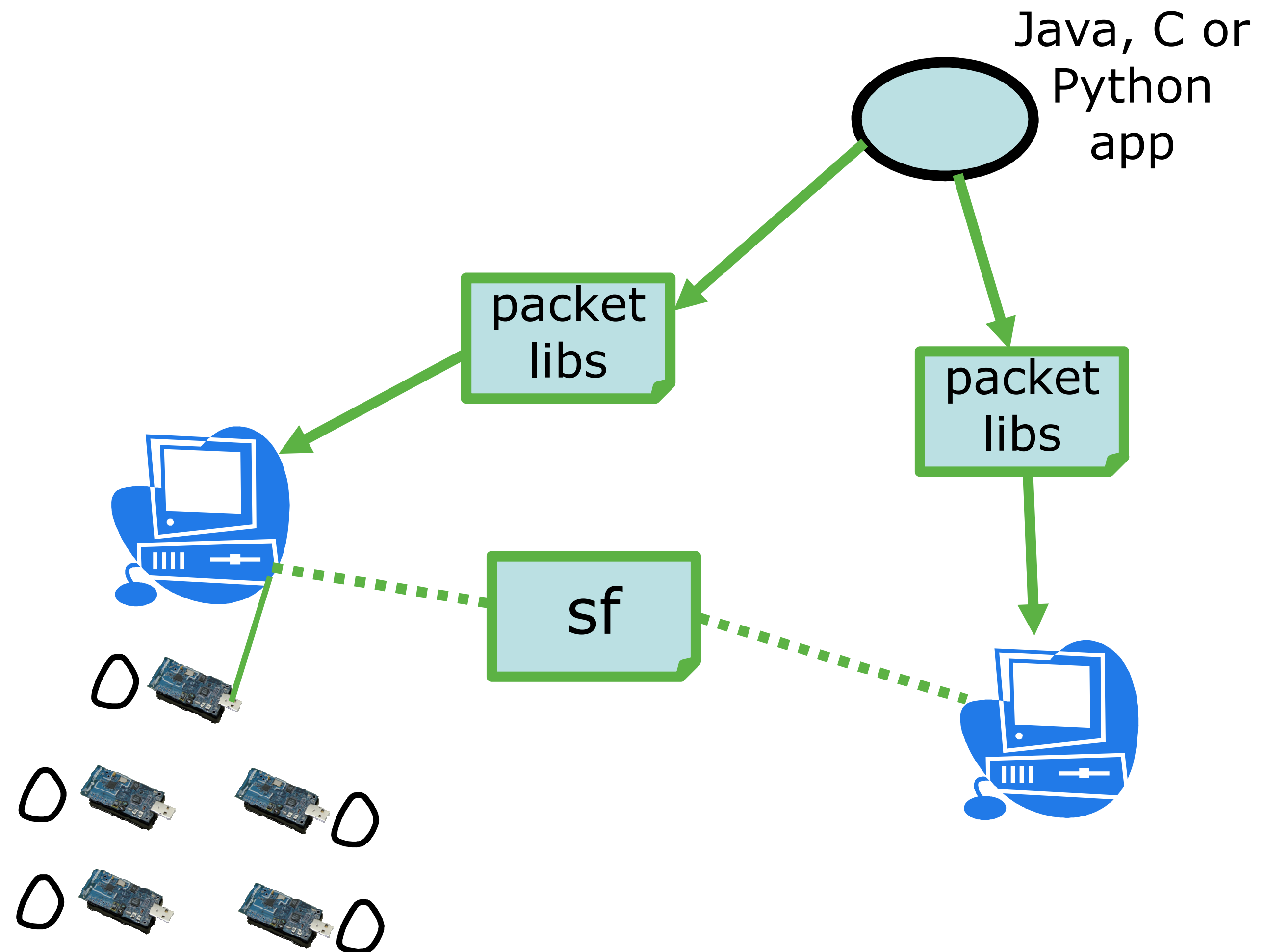
Build PC Applications



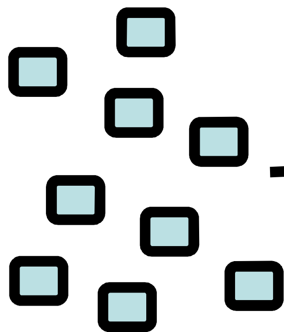
PC Applications: Extracting Information from TinyOS

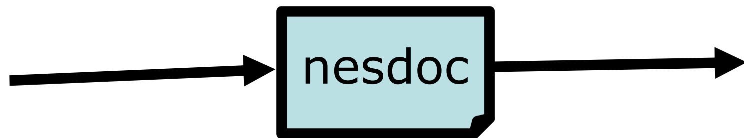


PC Applications: Talking to Motes



Document TinyOS





Interfaces and components

http://www.tinyos.net/tinyos-2.x/doc

tos.lib.ftsp
tos.lib.net
tos.lib.net.4bitle
tos.lib.net.6lowpan
tos.lib.net.ctp
tos.lib.net.drip
tos.lib.net.lqi
tos.lib.power
tos.lib.printf
tos.lib.serial
tos.lib.timer

tos.system

Components

ActiveMessageAddr
AMQueueEntryP
AMQueueImplP
AMQueueP
AMReceiverC
AMSenderC
AMSnooperC
ArbiterP
ArbitratedReadStrea
BitVectorC
FcfsArbiterC
FcfsResourceQueue
LedsC
LedsP
LocalTimeMilliC
MainC
NoLedsC
PoolC
PoolP
QueueC
RandomC
RandomMlcgC
RealMainP
RoundRobinArbiter
RoundRobinResourc
SchedulerBasicP
SimpleArbiterP
SimpleRoundRobin

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
Leds
LedsP
GeneralIO
GeneralIO
GeneralIO
Init
PlatformLedsC

http://www.tinyos.net/tinyos-2.x/doc/nesdoc/telosa/chtml/tos.system.LedsC.html

TOSSIM

Răzvan Musăloiu-E. (JHU)

What is TOSSIM?

Discrete event simulator

ns2

Alternatives

Cycle-accurate simulators

Avrora, MSPSim

Two directions

Port

make PC a supported platform

TOSSIM
in tinyos-1.x

Virtualize

simulate one of the supported platforms

TOSSIM
in tinyos-2.x

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

TOSSIM

```
tos/lib/tossim
tos/chips/atm128/sim
tos/chips/atm128/pins/sim
tos/chips/atm128/timer/sim
tos/chips/atm128/spi/sim
tos/platforms/mica/sim
tos/platforms/micaz/sim
tos/platforms/micaz/chips/cc2420/sim
```

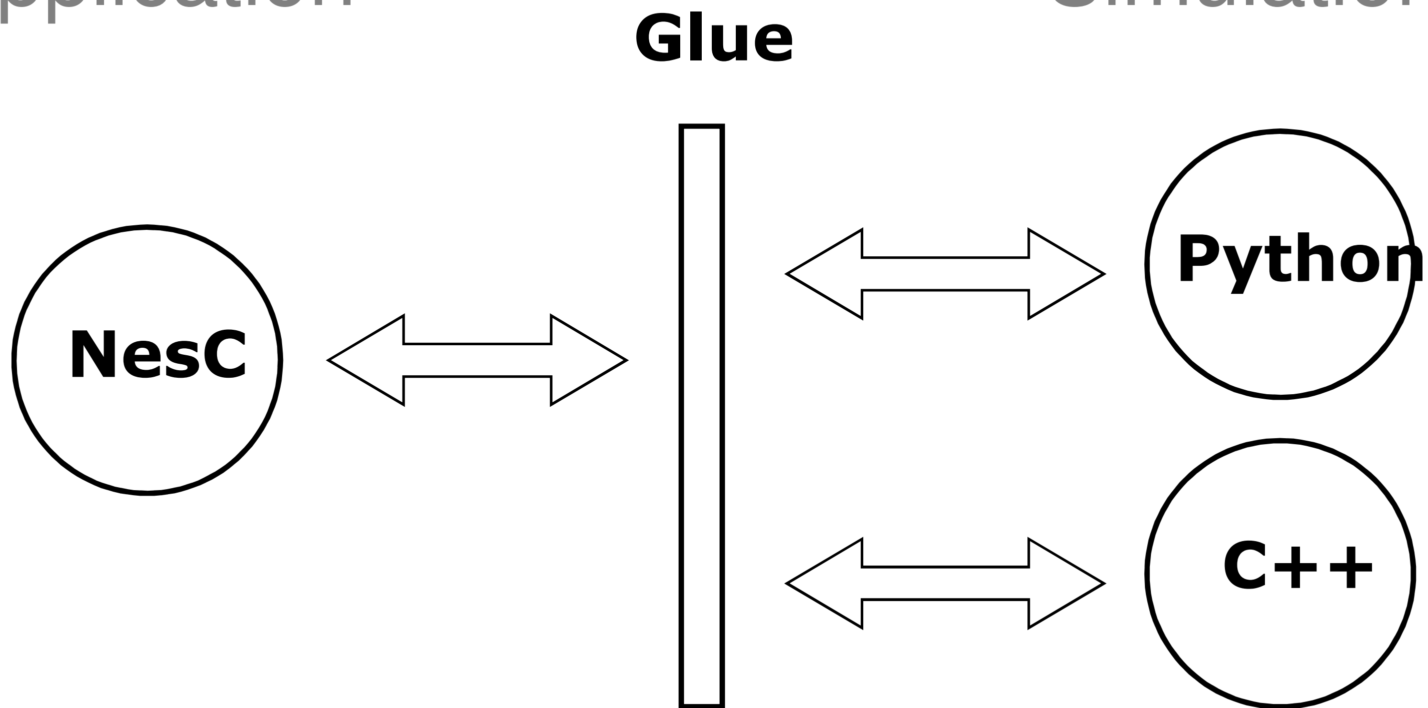
Application

```
Makefile
*.nc
*.h
```

Simulation Driver

```
*.py | *.cc
```

Quick Overview



The Building Process

```
$ make micaz sim
```

1. Generate an XML schema — *app.xml*
2. Compile the application — *sim.o*
3. Compile the Python support — *pytossim.o*, *tossim.o*, *c-support.o*
4. Build a share object — *_TOSSIMmodule.o*
5. Copying the Python support — *TOSSIM.py*

```
$ ./sim.py
```

TOSSIM.py

Tossim

Radio

Mote

Packet

Mac

TOSSIM.Tossim

.getNode() → TOSSIM.Mote

.radio() → TOSSIM.Radio

.newPacket() → TOSSIM.Packet

.mac() → TOSSIM.Mac

.runNextEvent()

.ticksPerSecond()

.time()

Simulating 10 seconds

```
from TOSSIM import *
t = Tossim([])
  ...
while t.time() < 10*t.ticksPerSecond():
    t.runNextEvent()
```

dbg

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

*char**	sim_time_string()
sim_time_t	sim_time()
int	sim_random()
sim_time_t	sim_ticks_per_sec()

typedef *long long int sim_time_t*;

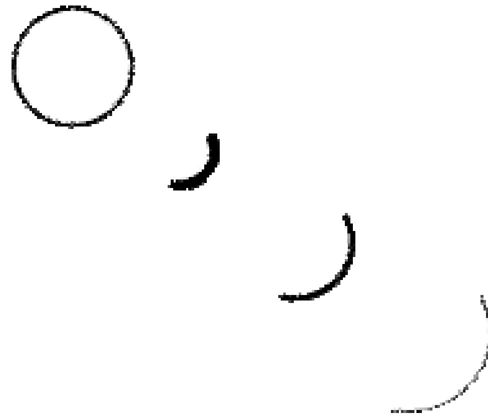
Radio Model

Closest-fit Pattern Matching (CPM)

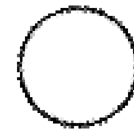
Improving Wireless Simulation Through Noise Modeling
HyungJune Lee, Alberto Cerpa, and Philip Levis
IPSN 2007

Radio Model

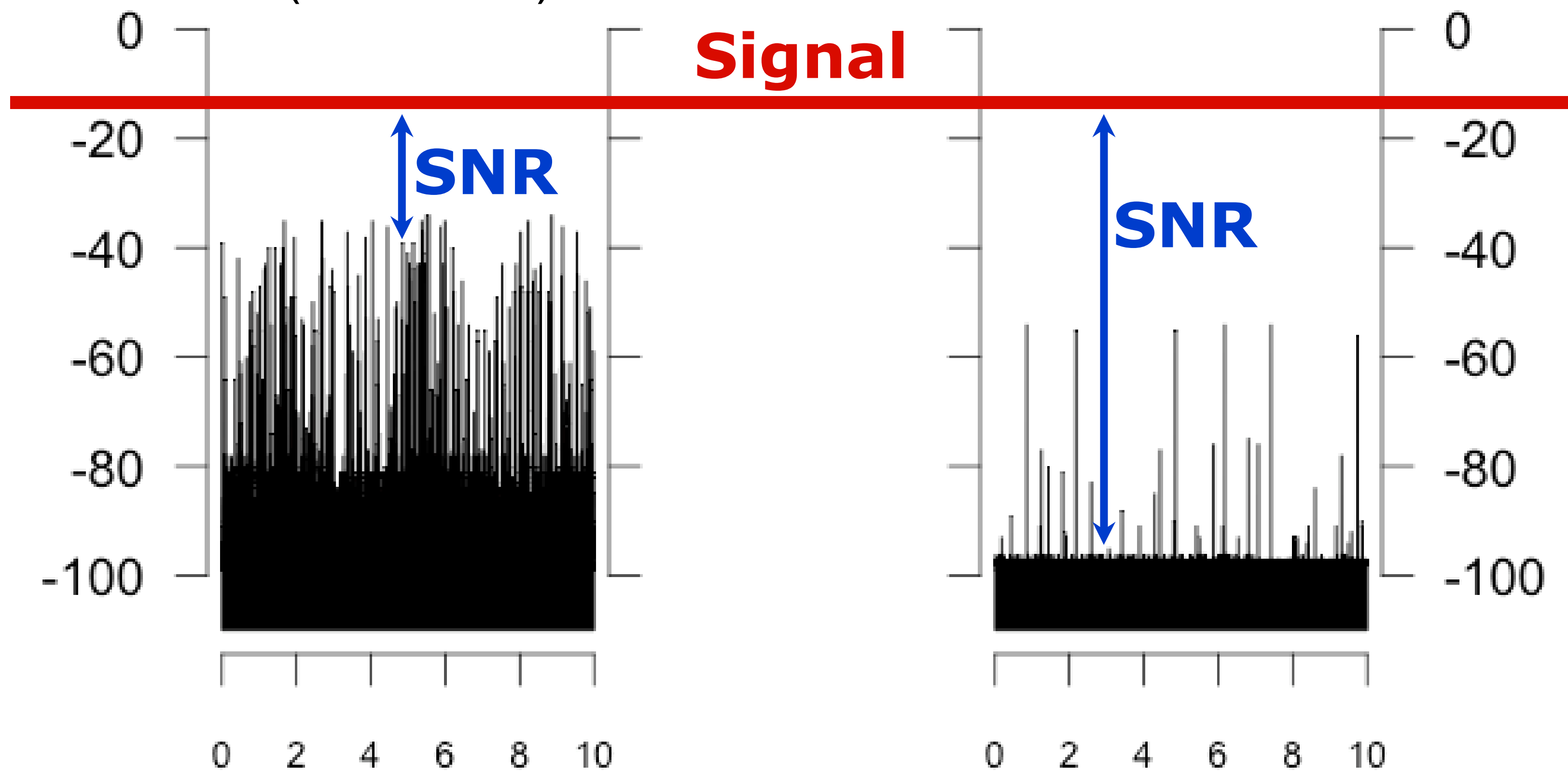
Sender



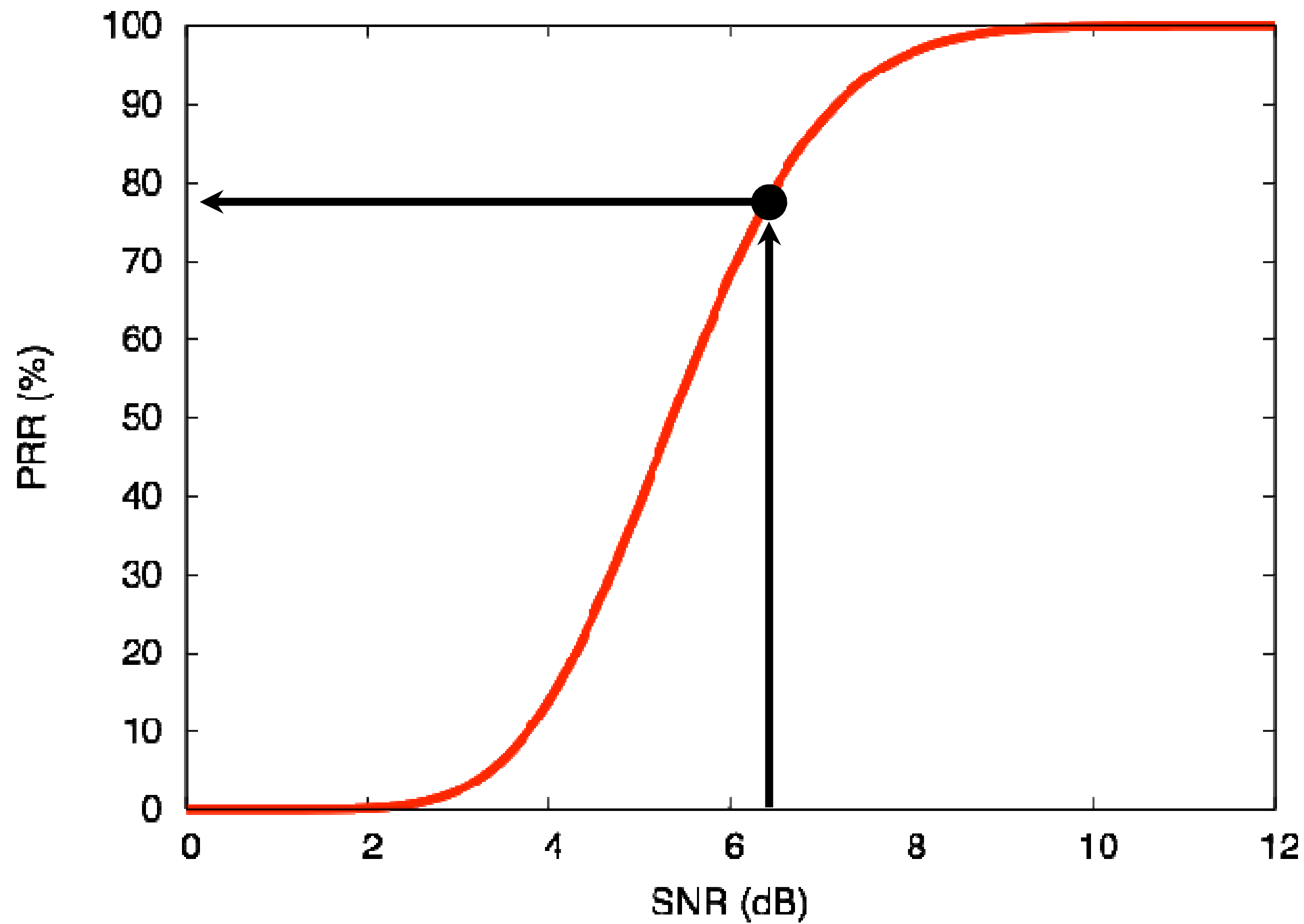
Receiver



Noise Level



CC2420 SNR/PRR



TOSSIM.Radio

.add(*source, destination, gain*)

.connected(*source, destination*) → True/False

.gain(*source, destination*)

TOSSIM.Mote

.bootAtTime(*time*)

.addNoiseTraceReading(*noise*)

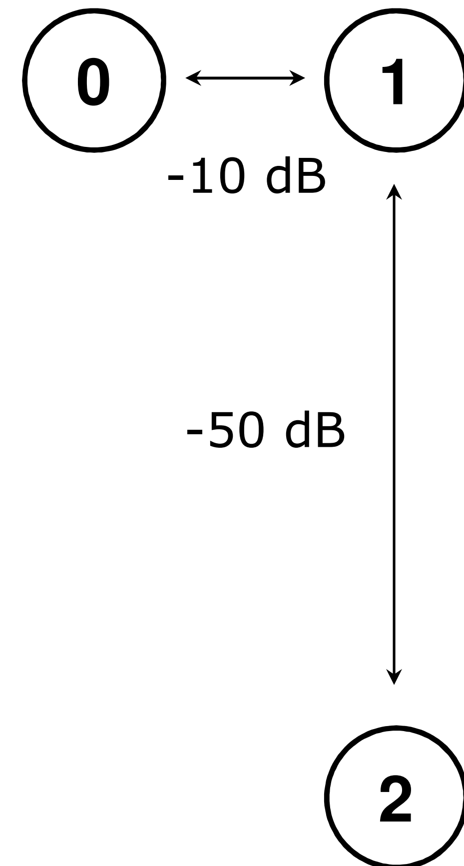
.createNoiseModel()

.isOn() → True/False

.turnOn()/.turnOff()

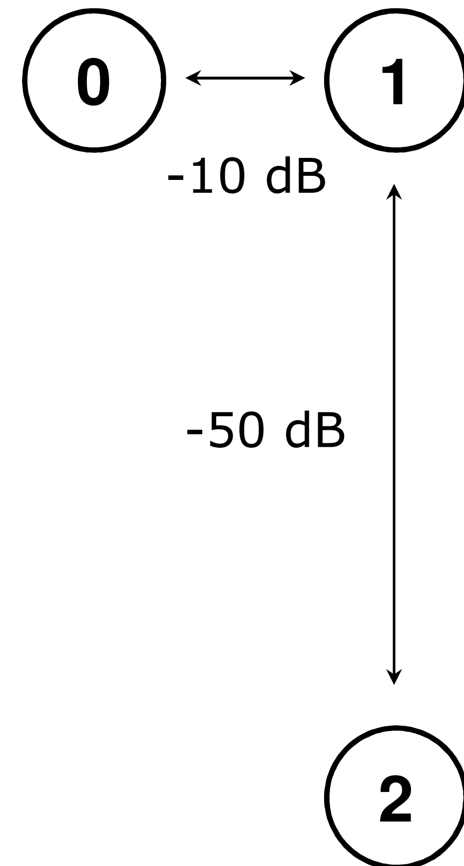
Example

```
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)

```
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

FLIDs

- Default behavior on safety violation is to output a FLID (Fault Location IDentifier) 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

Decoding a FLID

```
$ tos-decode-flid ./build/micaz/flids.txt 00001020
Deputy error message for flid 0x0048:

BlinkFailC__a <= BlinkFailC__a + BlinkFailC__i++ + 1
   (with no overflow): BlinkFailC.nc:70:
Assertion failed in CPtrArithAccess: BlinkFailC__a +
   BlinkFailC__i++ + 1 <= BlinkFailC__a + 10 (with no
   overflow)
```

Safe Components

- Safety is “opt in” at the level of nesC components
- This component is compiled as safe code:

```
generic module SimpleArbiterP() @safe() { … }
```

- These components are “trusted” code:

```
generic module SimpleArbiterP() @unsafe() { … }
generic module SimpleArbiterP() { … }
```

- 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`**

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

- Or if **`msg`** may be null

```
message_t* ONE_NOK msg;
```

- Most annotations have a **`_NOK`** form
  - But avoid using it when possible

Annotation 2

- 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`**

```
message_t* ONE uartQueue[10];
```

Annotation 3

- To declare **`reqBuf`** as a pointer that always points to a valid block of at least **`reqBytes`** **`uint8_t`**s:

```
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

Annotation 4

- Multiple-indirect pointers require an annotation at each level:

```
int *ONE *ONE pp = ...;
```

- 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

```
cc2420_header_t* ONE x = TCAST(
  cc2420_header_t* ONE,
  (uint8_t *)msg +
  offsetof(message_t, data) -
  sizeof(cc2420_header_t)
);
```

Interface Annotation 1

- The **getPayload()** command from the Packet interface might be annotated like this:

```
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:

```
* @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

Threads

Kevin Klues (UCB)

The Great Divide

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

```
void myFunc() {
    error_t e = read();
    //continue execution flow
}
void readDone(uint8_t val, error_t e) {
    //read() continuation code
}
```

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

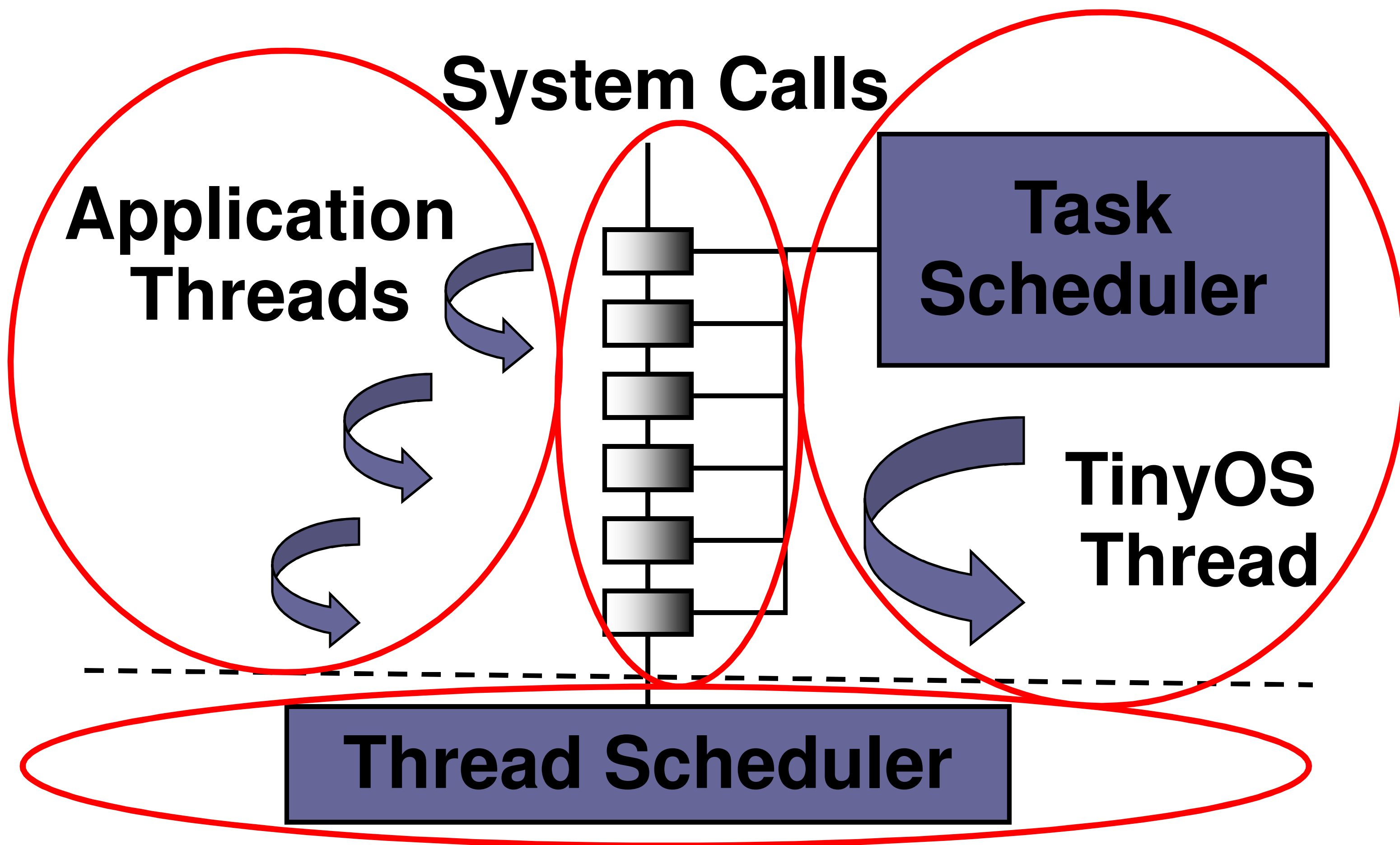
```
void myFunc() {
    error_t e;
    uint8_t val = read(&e);
    //read() continuation code
}
```

TOSThreads aims to resolve this fundamental tension

TOSThreads 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



Blink Example (nesC)

```
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)

```
#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

```
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

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

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

Boot Sequence

Thread Scheduler Boot

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

New Main

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

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

Task Scheduler

Original

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

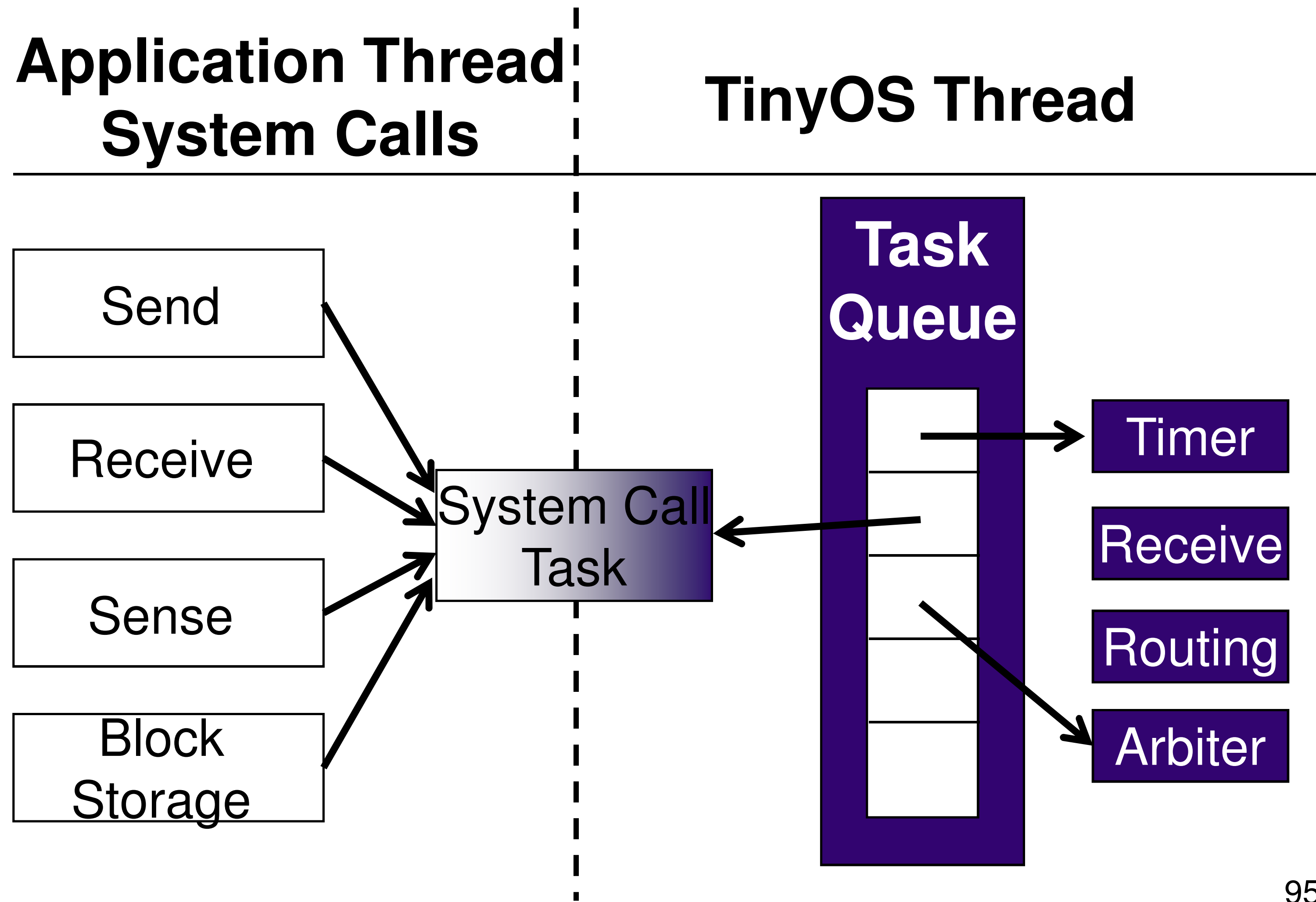
New

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

Interrupt Handlers

```
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



Resources

- TOSThreads Tutorial

  http://docs.tinyos.net/index.php/TOSThreads_Tutorial

- TOSThreads TEP

  http://www.tinyos.net/tinyos-2.x/doc/html/tep134.html

- Source Code

  System code: tinyos-2.x/tos/lib/tosthreads

  Example Applications: tinyos-2.x/apps/tosthreads

Protocols

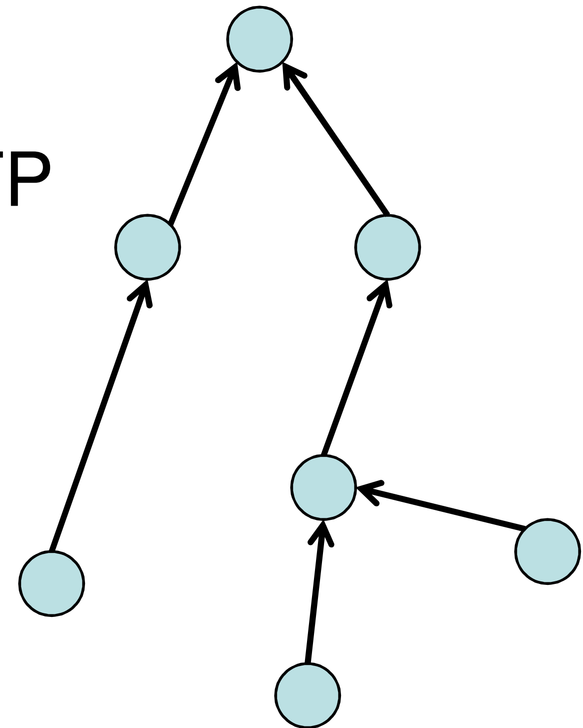
Omprakash Gnawali (USC)

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



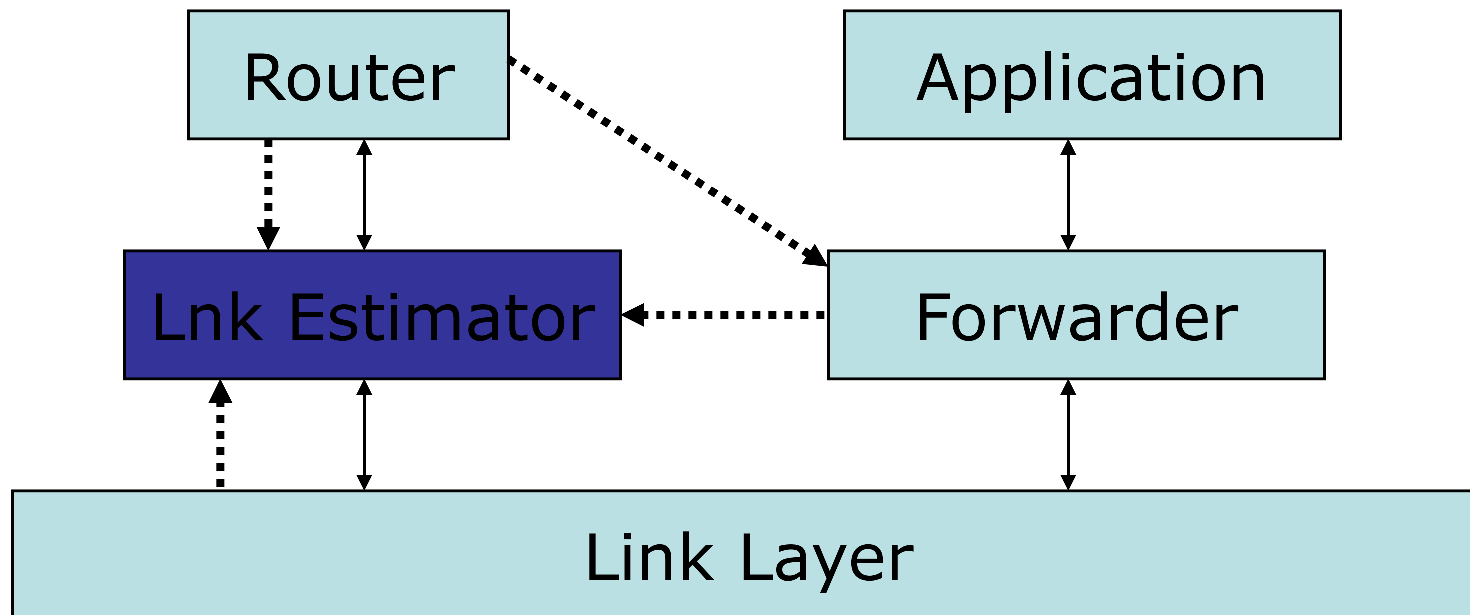
MultihopLQI

- Mostly tested and used on platforms with CC2420
  - MicaZ, TelosB, …
- Small code footprint
- tos/lib/net/lqi

CTP

- Platform independent
- More consistent performance than with MultihopLQI
- Code footprint can be a concern
- tos/lib/net/ctp

CTP Architecture



CTP Link Estimator

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

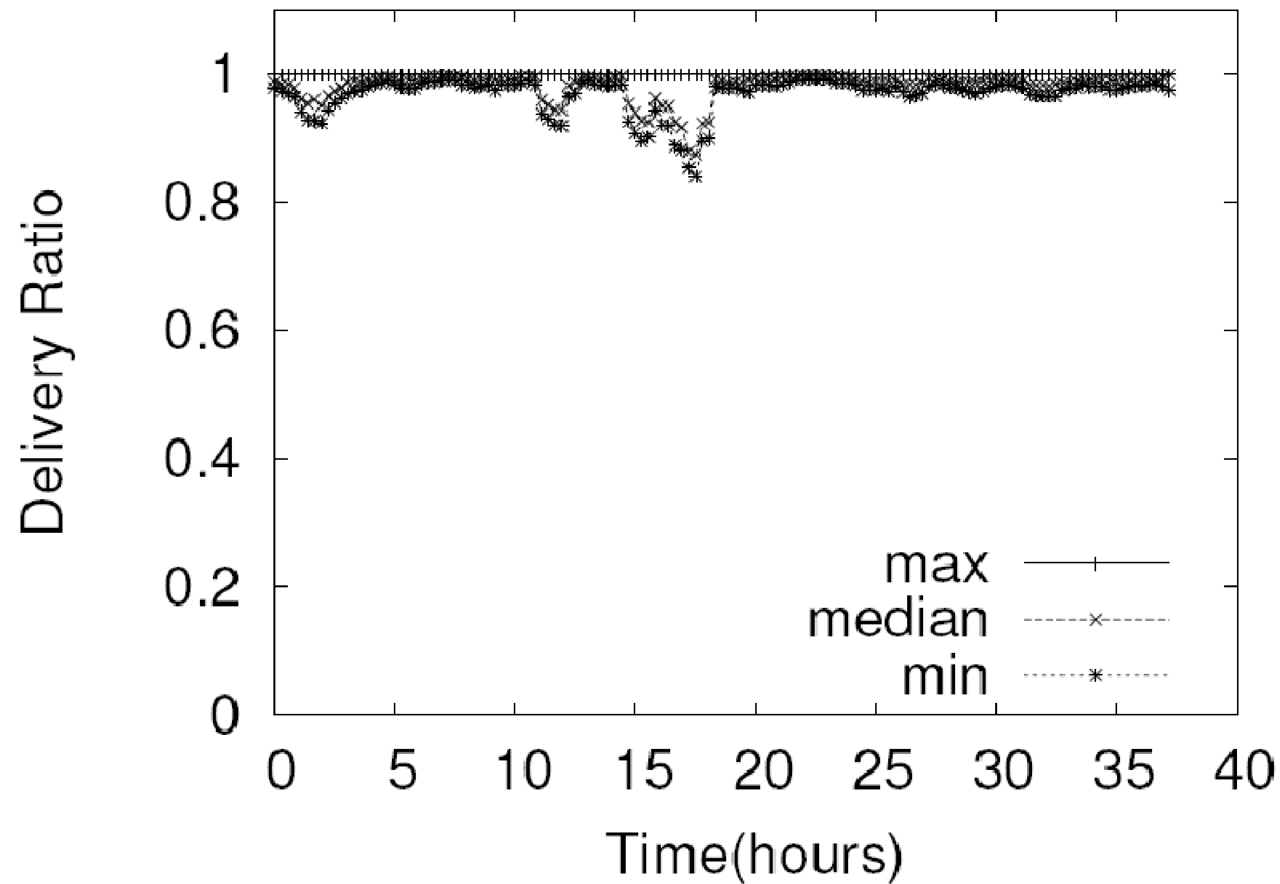
CTP Router

- ETX path metric
- Beacon interval can be 64 ms-x mins
- Select new path if better by at least 1.5 ETX
- Alternate parents

CTP Forwarder

- Duplicate suppression
- Retransmissions
- Loops trigger route updates
- Forward through alternate parents

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/index.php/TinyOS_Tutorials
  - Note: information at the following site may well be helpful too
    - http://www.tinyos.net/tinyos-2.x/doc/
- Lab#0 (mandatory)
  - 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/