TinyOS 2.0: A wireless sensor network operating system

David Gay, Intel Research Berkeley

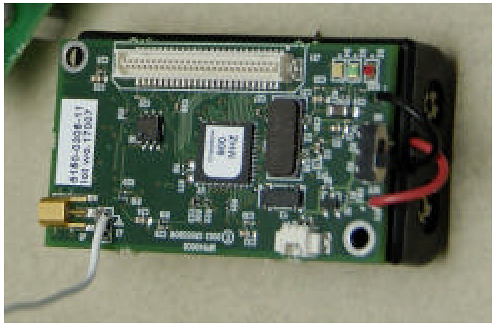
with Phil Levis, Vlado Handziski, Jonathan Hui,
Jan-Hinrich Hauer, Ben Greenstein, Martin Turon,
Kevin Klues, Cory Sharp, Robert Szewczyk,
Joe Polastre, David Moss, Henri Dubois-Ferrière,
Gilman Tolle, Philip Buonadonna, Lama Nachman,
Adam Wolisz and David Culler

Sensor Networks

Sensor network are collections of small, battery operated computers with

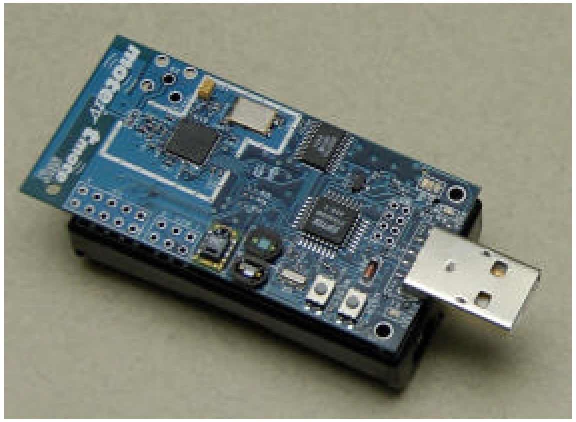
- sensors, and possibly actuators, to sense and control their environment
- radios, to report data and receive instructions
- typical expected lifetimes range from a few months to several years

Some Typical Devices



mica2 (2002)

- 8MHz ATmega128
- 4kB RAM, 128kB flash
- 512kB external flash
- 20kb/s custom radio
- many different sensor boards
- 2 AA batteries:
  - radio+cpu: 75mW
  - sleep mode: 140µW



telosb (2004)

- 1MHz TI MSP430
- 10kB RAM, 48kB flash
- 512kB external flash
- 250kb/s 802.15.4 radio
- built-in sensors
- 2 AA batteries:
  - radio+cpu mode: 63mW
  - sleep mode: 30 µW

lifetime: a few days to several years

Sensor Networks

Sensor network are collections of small, battery operated computers with

- sensors, and possibly actuators, to sense and control their environment
- radios, to report data and receive instructions
- typical expected lifetimes range from a few months to several years

Suggested applications include:

- data collection, environmental or industrial monitoring, object tracking

Today:

- We'll build a simple "anti-theft" application using TinyOS 2.0, which
  - detects theft by light level or movement
  - reports theft by blinking, beeping, to neighbours or to a central server
  - is configurable from a central server

  in less than 200 lines of code

Challenges

Driven by interaction with environment (“Am I being stolen?”)

- Data collection and control, not general purpose computation
- Requires event-driven execution

Extremely limited resources (“2 AA’s, 4kB of RAM”)

- Very low cost, size, and power consumption

Reliability for long-lived applications (“Don’t steal me in a year!”)

- Apps run for months/years without human intervention
- Reduce run time errors and complexity

Real-time requirements (“What is movement anyway?”)

- Some time-critical tasks (sensor acquisition and radio timing)
- Timing constraints through complete control over app and OS

Constant hardware evolution

Outline

TinyOS and nesC overview

Building a simple anti-theft application

- The Basics
- “Advanced” Networking
- “Basic” Networking

  BREAK
- Managing Power
- For experts: implementing device drivers
  - resource and power management
  - low-level code and concurrency

Review and Conclusion

TinyOS and nesC

TinyOS is an operating system designed to target limited-resource sensor network nodes

- TinyOS 0.4, 0.6 (2000-2001)
- TinyOS 1.0 (2002): first nesC version
- TinyOS 1.1 (2003): reliability improvements, many new services
- TinyOS 2.0 (2006): complete rewrite, improved design, portability, reliability and documentation

TinyOS and its application are implemented in nesC, a C dialect:

- nesC 1.0 (2002): Component-based programming
- nesC 1.1 (2003): Concurrency support
- nesC 1.2 (2005): Generic components, “external” types

TinyOS in a nutshell

System runs a single application

- OS services can be tailored to the application’s needs

These OS services include

- timers, radio, serial port, A/D conversion, sensing, storage, multihop collection and dissemination, …

Application and services are built as

- a set of interacting components (as opposed to threads)
- using a strictly non-blocking execution model
  - event-driven execution, most service requests are split-phase

Implementation based on a set of OS abstractions

- tasks, atomic with respect to each other; interrupt handlers
- resource sharing and virtualisation, power management
- hardware abstraction architecture

nesC in a seashell

C dialect

Component based

- all interaction via interfaces
- connections ("wiring") specified at compile-time
- generic components, interfaces for code reuse, simpler programming

"External" types to simplify interoperable networking

Reduced expressivity

- no dynamic allocation
- no function pointers

Supports TinyOS's concurrency model

- must declare code that can run in interrupts
- atomic statements to deal with data accessed by interrupts
- data race detection to detect (some) concurrency bugs

The Basics

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.
  - Theft detection algorithm: 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 C
  uses interface Read<uint16_t>;
}
implementation
  event void Boo
    call Check.sta
  }
  event void Che
    call Read.read();
  }
  event void Read.readDone(er
    if (ok == SUCCESS && val <
      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();
}
```

Programs are built out of named components
A component provides and uses interfaces
Interfaces contain commands and events, which are just functions
A module is a component implemented in C

The Basics – Interfaces

```
module AntiTheftC {
  uses interface Boot;
  uses interface Timer<TMilli> as
  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();
  }
}
```

Interfaces specify the interaction between two components, the *provider* and the *user*. This interaction is just a function call.
commands are calls from user to provider
events are calls from provider to user

```
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();
}
```

The Basics – Interfaces and 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(
  }
  event void Check.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t ok, uint16_t val) {
    if (ok == SUCCESS && val < 200)
      theftLed();
  }
}
```

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 – Interfaces and 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(
  }
  event void Check.fired()
    call Read.read();
  }
  event void Read.readDone(error_t ok, uint16_t val) {
    if (ok == SUCCESS && val < 200)
      theftLed();
  }
}
```

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

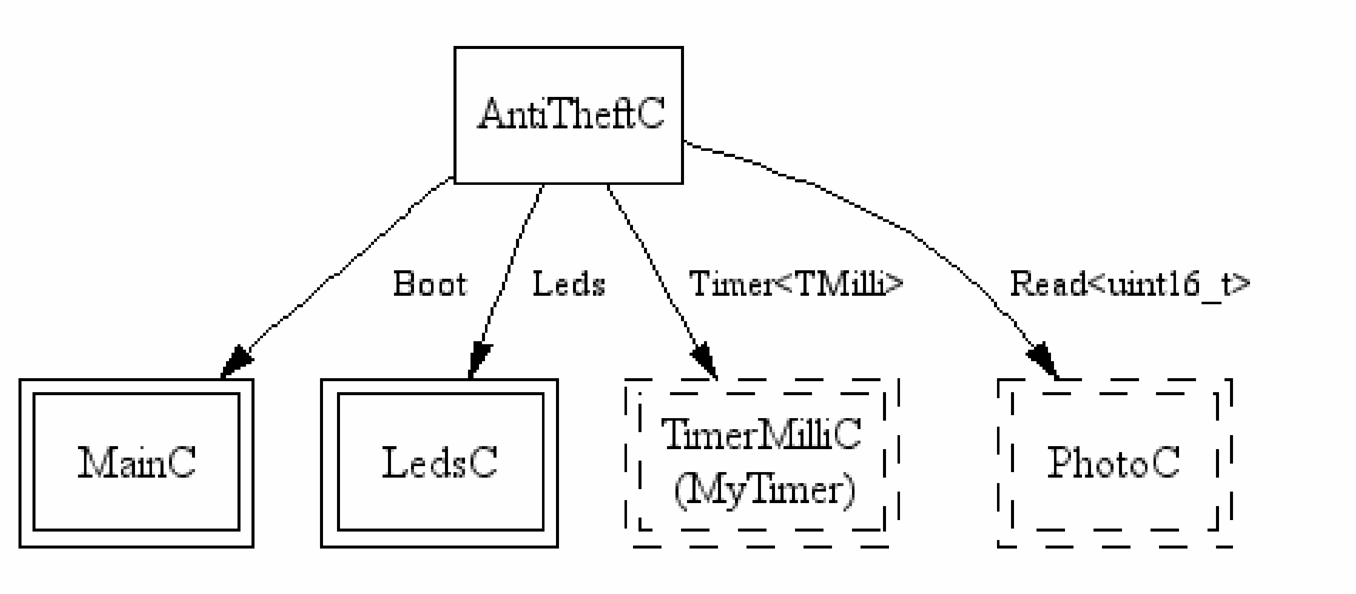
```
configuration AntiTheftAppC { }
implementation
{
  components AntiT

  AntiTheftC.Boot -
  AntiTheftC.Le

  components
  AntiTheftC.Ch

  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

The Basics


AntiTheftC
Boot
Leds
Timer<TMilli>
Read<uint16_t>
MainC
LedsC
TimerMilliC
(MyTimer)
PhotoC

The Basics

Let’s improve our anti-theft device. A clever thief could still steal our motes by keeping a light shining on them!

- But, however clever, 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

The Basics – 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.rea
  if (ok == SUCCESS)
    post checkAcceleration()
}

task void checkAcceleratio
    ... 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

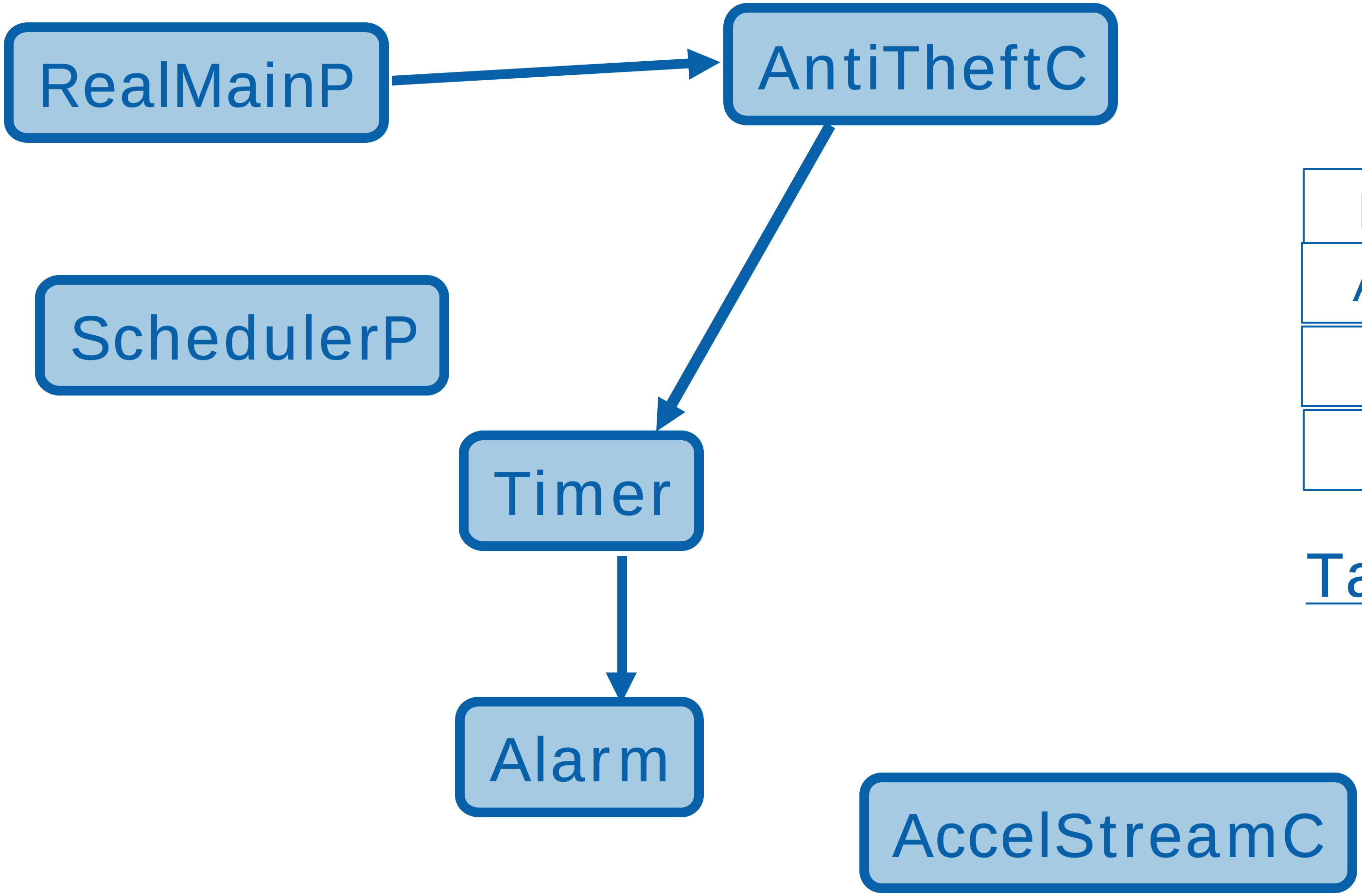
```
interface ReadStream<val_t> {
  command error_t postBuffer(val_t* buf, uint16_t count);
  command error_t read(uint32_t period);
  event void readDone(error_t ok, uint32_t actualPeriod);
}
```

The Basics – 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 < S
     {
       int16_t diff = accelSa
       var += diff * diff;
     }
   if (var > 4 * SAMPLES) t
}
```

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.

The Basics - TinyOS Execution Model



RealMainP

AntiTheftC

SchedulerP

Timer

Alarm

AccelStreamC

Stack

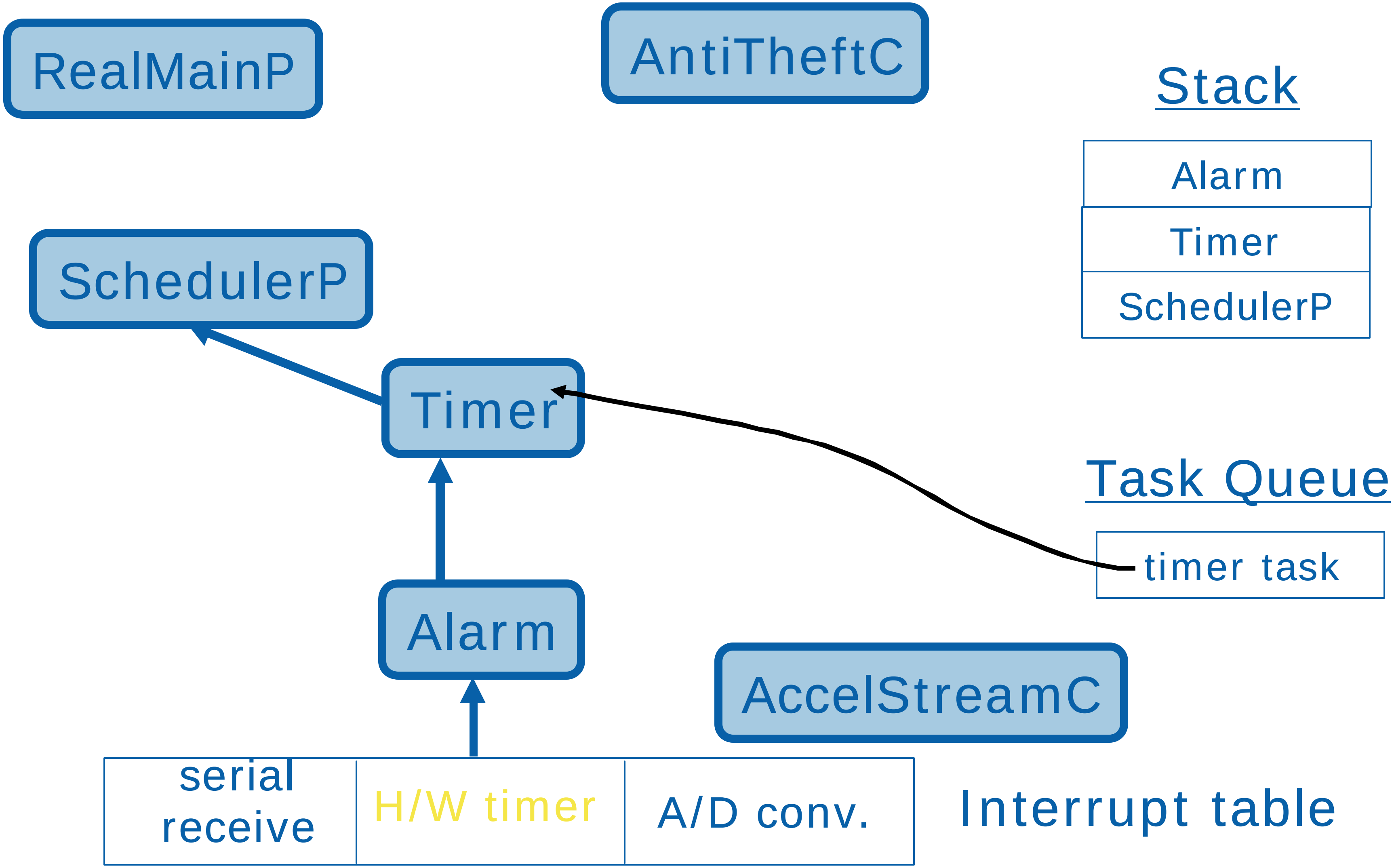
RealMainP
AntiTheftC
Timer
Alarm

Task Queue

serial receive	H/W timer	A/D conv.

Interrupt table

The Basics - TinyOS Execution Model



RealMainP

AntiTheftC

SchedulerP

Timer

Alarm

AccelStreamC

Stack

Alarm

Timer

SchedulerP

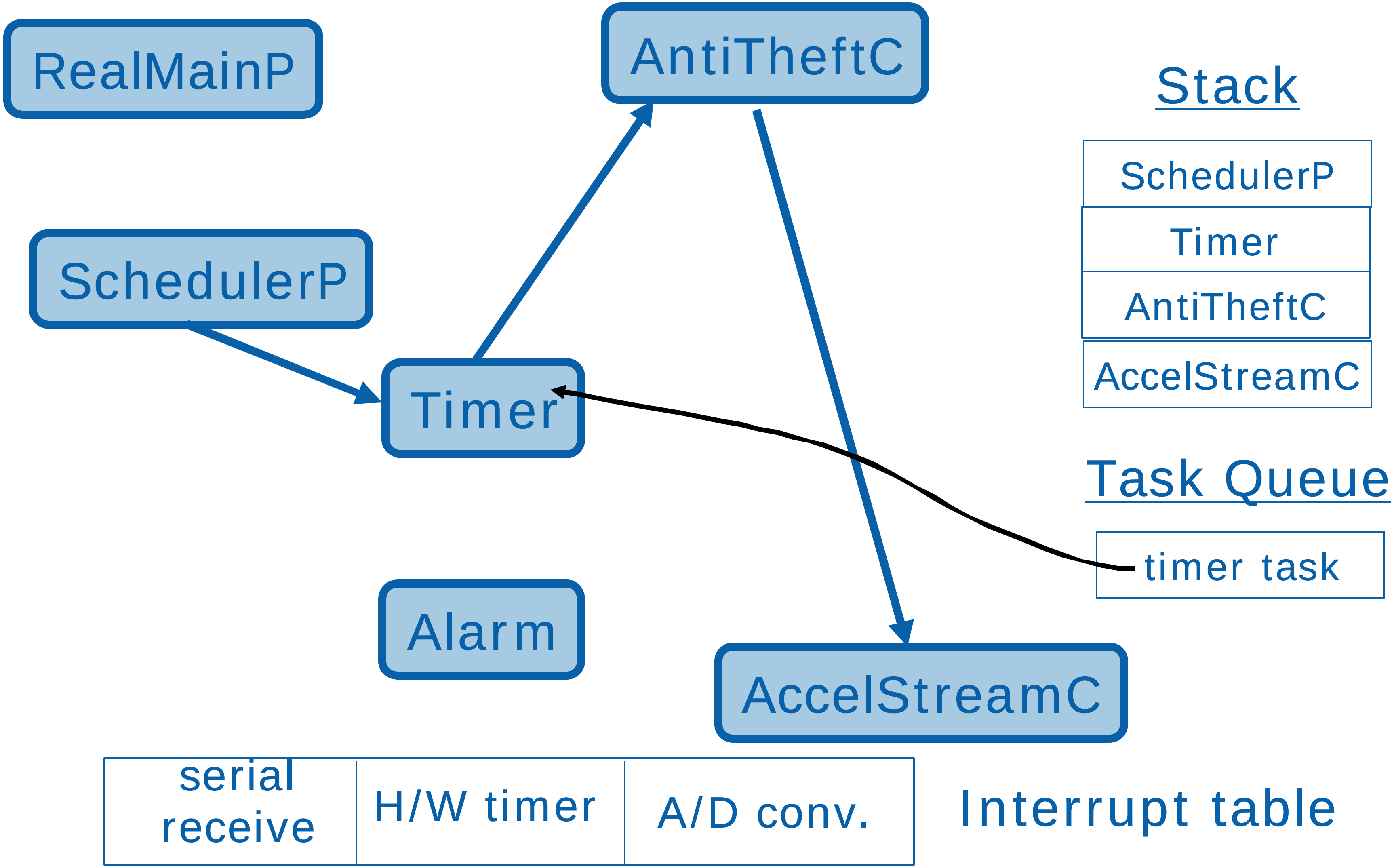
Task Queue

timer task

serial receive	H/W timer	A/D conv.

Interrupt table

The Basics - TinyOS Execution Model



RealMainP

AntiTheftC

SchedulerP

Timer

Alarm

AccelStreamC

Stack

- SchedulerP
- Timer
- AntiTheftC
- AccelStreamC

Task Queue

- timer task

serial receive	H/W timer	A/D conv.

Interrupt table

The Basics - 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
- Components interact via interfaces only

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)

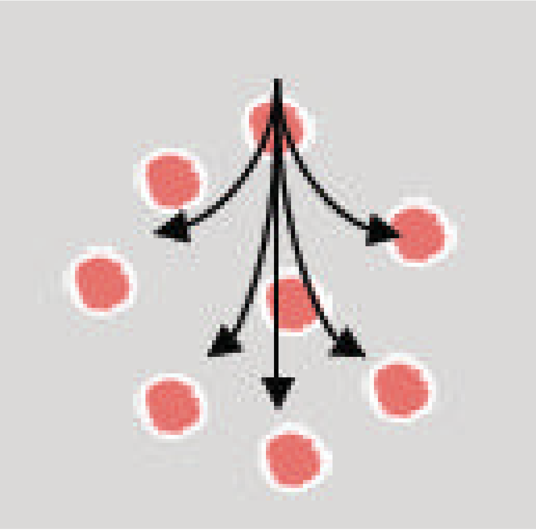
- (Mostly) represented by instantiatable generic components
  - This instantiation happens at compile-time! (think C++ templates)
- All slow system requests are split-phase

"Advanced" Networking

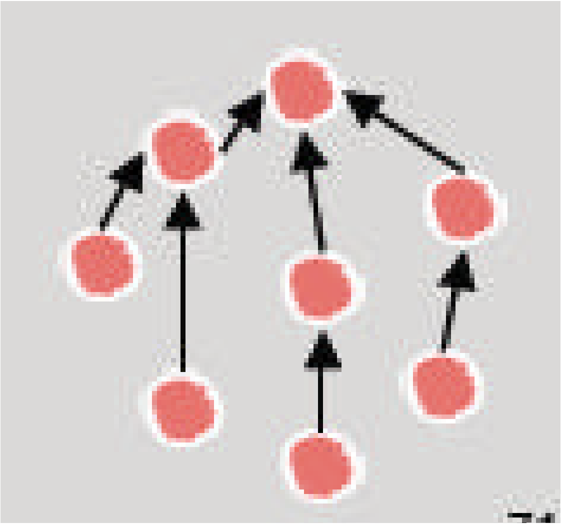
TinyOS 2.0 contains two useful network protocols:

- dissemination, that disseminates a value to all nodes in the network
  - use for reconfiguration, program dissemination, i.e., network control
- collection, that allows all nodes to report values to *root* nodes
  - the simplest data collection mechanism

Dissemination



Collection



“Advanced” Networking

Different anti-theft mechanisms may be appropriate for different times or places. Our perfect anti-theft system must be configurable!

- We want to send some settings to all motes selecting between dark and acceleration detection.
- We’ll also allow selection of siren-based alerts instead of the “bright flashing light”, and of the theft check interval
- We’ll use the dissemination protocol to achieve this

What we’ll see:

- How to use the dissemination protocol
- How to start (and stop) services
- “External” types, and their use in networking

“Advanced” Networking – “External” Types

```
#include “antitheft.h”
module AntiTheftC {
 ... uses interface DisseminationValue<settings_t> as SettingsValue;
} implementation {
 settings_t settings;
 event void SettingsValue.changed() {
                                   SettingsValue.get();
                                   ect;

                                   ->checkInterval);

   if (settings.detect & DETE
      call ReadStream.postB
      call ReadStream.read(
   }
 }
```

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

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

“Advanced” Networking – Dissemination

```
#include “antitheft.h”
module AntiTheftC {
 ... uses interface DisseminationValue<settings_t> as SettingsValue;
} implementation {
 settings_t settings;
 event void SettingsValue.changed() {
   const settings_t *newSettings = c
   settings.detect = newSettings->de
   settings.alert = newSettings->aler
   call Check.startPeriod(newSetting
 }

 event void Timer.fired() {
   if (settings.detect & DETECT_DARK)
     call Read.read();
   if (settings.detect & DETECT
       call ReadStream.postBuf
       call ReadStream.read(10
   }
 }
```

```
interface DisseminationValue<t> {
  command const t* get();
  event void changed();
}
```

Dissemination is simple to use:

- The changed event is signalled when new settings are received
- The get command is used to obtain the received value

We can then simply read the received settings

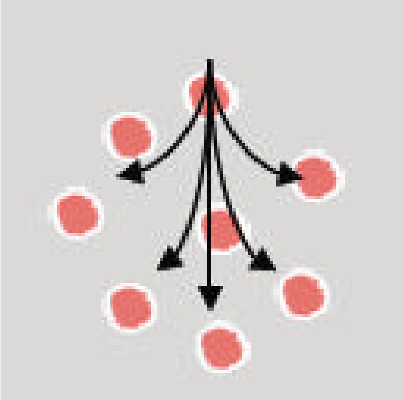
“Advanced” Networking – Dissemination

```
configuration AntiTheftAppC { }
implementation
{
  ...
  components ActiveMessageC,
    new DisseminatorC(settings_t, DIS_SETTINGS);
  AntiTheftC.SettingsValue -> DisseminatorC;
  AntiTheftC.RadioControl -> ActiveMessageC;
}
```

Finally, we need to wire in the new functionality:
- We create a disseminator for the settings_t type
- We wire ActiveMessageC to start to the radio

(we’ll see why in a little bit)

Dissemination – How does it work?



Use local broadcasts and packet suppression

- Scale to a wide range of densities
- Control transmissions over *space*

100% eventual reliability

- Disconnection, repopulation, etc.
- Continuous process

Maintenance: exchange metadata (e.g., version numbers, hashes) at a low rate to ensure network is up to date

Propagation: when a node detects an inconsistency, the network quickly broadcasts the new data

[Slide Courtesy Phil Levis]

“Advanced” Networking – Starting Services

```
uses interface SplitControl as RadioControl;
...
event void Boot.booted() {
  call Check.startPeriodic(1000);
  call RadioControl.start();
}

event void RadioControl.startDone(
event void RadioControl.stopDone(
```

```
interface SplitControl {
  command error_t start();
  event void startDone(error_t ok);

  command error_t stop();
  event void stopDone(error_t ok);
}
```

Whenever possible, TinyOS 2.0, starts and stops services automatically.
This isn’t possible for the radio (no knowledge of when messages might arrive), so responsibility passed to the programmer.
Must turn on radio for dissemination service to work.
SplitControl is one of the interfaces for starting and stopping services

- Split-phase, used when start/stop may take a while

“Advanced” Networking - Collection

What if thieves aren’t deterred by sirens and flashing lights?

We need to report the theft!

We’ll use the tree-based collection to send theft reports to a base station.

What we’ll see:

- collection protocol
- message_t, TinyOS’s message buffer type
- Send, TinyOS’s address-less send interface

“Advanced” Networking - Collection

```
interface Send as AlertRoot;
interface StdControl as CollectionControl;
...
message_t alertMsg;
event void RadioControl.startDone(error_t ok) {
   if (ok == SUCCESS) call CollectionControl.start();
}
void theft() {
  if (settings.alert & ALERT_LEDS)
    theftLed();
  if (settings.alert & ALERT_ROOT)
    {
      alert_t *newAlert = ca
      newAlert->stolenId =
      call AlertRoot.send(&al
    }
}
event void AlertRoot.sendD
```

```
interface StdControl {
  command error_t start();
  command error_t stop();
}
```

Before we can report anything, we need to:
- Start the radio (already done)
- Start the collection service

“Advanced” Networking - Collection

```
interface Send as AlertRoot;
interface StdControl as CollectionControl;
...
message_t alertM
event void RadioC
   if (ok == SUCC
}
void theft() {
  if (settings.alert
    theftLed();
  if (settings.alert & ALERT_ROOT)
    {
      alert_t *newAlert = call AlertRoot.getPayload(&alertMsg);
      newAlert->stolenId
      call AlertRoot.send(&
    }
}
event void AlertRoot.sendDone(message_t *msg, error_t ok) { }
```

```
interface Send {
 command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t ok);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);
}
```

Collection messages are sent
- By placing data in a *message_t* buffer
- Using the Send interface

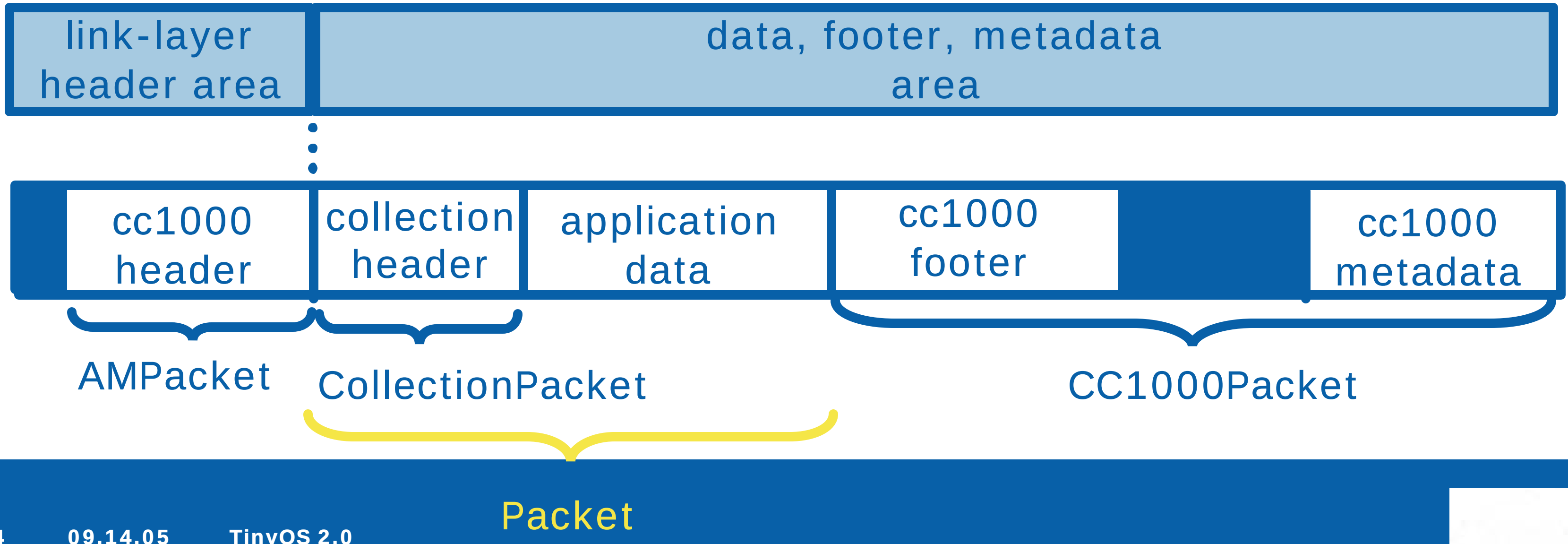
Networking: packet abstract data type

message_t is a platform-defined type for holding packets

- a fixed size byte array
- capable of holding MTU of all data-link layers (platform-selected)

accessed **only** via interfaces:

- Packet: general payload access, provided at each layer
- *xx*Packet: information for layer *xx*



link-layer header area

data, footer, metadata area

cc1000 header

collection header

application data

cc1000 footer

cc1000 metadata

AMPacket

CollectionPacket

CC1000Packet

Packet

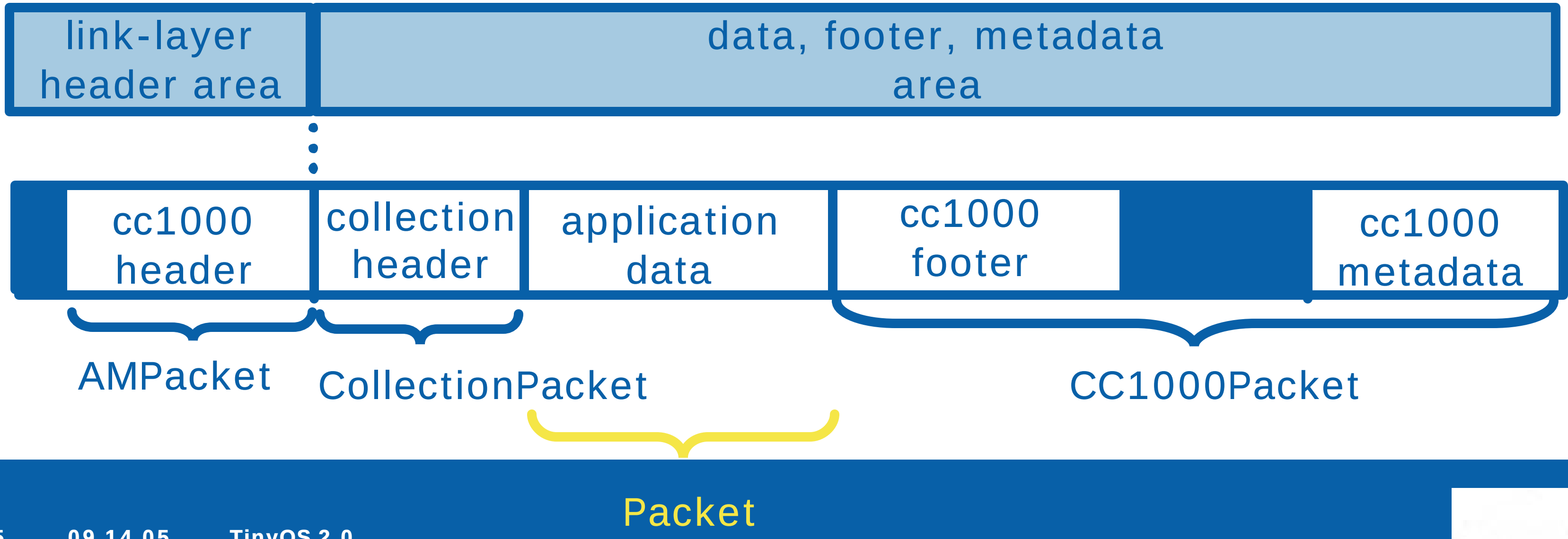
Networking: packet abstract data type

message_t is a platform-defined type for holding packets

- a fixed size byte array
- capable of holding MTU of all data-link layers (platform-selected)

accessed **only** via interfaces:

- Packet: general payload access, provided at each layer
- *xx*Packet: information for layer *xx*



link-layer header area

data, footer, metadata area

cc1000 header

collection header

application data

cc1000 footer

cc1000 metadata

AMPacket

CollectionPacket

CC1000Packet

Packet

“Advanced” Net

We use getPayload to get access to the portion of message_t available to the application

We use an external type to actually write the data

```
typedef nx_struct {
  nx_uint16_t stolenId;
} alert_t;
```

```
interface Send {
 command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t ok);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);
}
```

```
Ro
C
event void RadioC
   if (ok == SUCC
}
void theft() {
  if (settings.alert
    theftLed();
  if (settings.alert
    {
      alert_t *newAlert = call AlertRoot.getPayload(&alertMsg);
      newAlert->stolenId = TOS_NODE_ID;
      call AlertRoot.send(&alertMsg, sizeof *newAlert);
    }
}
event void AlertRoot.sendDone(message_t *msg, error_t ok) { }
```

“Basic” Networking

The police may not get there in time to catch the mote thief.

So, let’s alert the mote’s neighbours!

We’ll send a local broadcast message over the radio.

What we’ll see:

- active message-based single-hop messaging

“Basic” Networking - Interfaces

address-free interfaces for sending, receiving:

- Send: send a packet
- Receive: receive a packet

“active messages” interfaces:

- active messages has destination addresses
- active messages has “message type”, used for dispatch on reception
- AMSend: send a packet to an active message address
- Receive is reused
- Message type not specified in interfaces, but in configurations

```
interface AMSend {
 command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t ok);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg);
}
```

```
void theft() {
  ...
  if (settings.alert & ALERT_RADIO)
    call TheftSend.send(AM_BROADCAST_ADDR, &theftMsg, 0);
 }
event message_t *TheftReceive.receive
(message_t* msg, void *payload, uint8_t len) {
  theftLed();
  return msg;
 }
```

AMSend is just like send, but with a destination
The theft message has no data, so no use of the payload functions.

“Basic” Networking

```
interface Receive{
  event message_t* receive(message_t* msg, void* payload, uint8_t len);
  command uint8_t payloadLength();
  command void* getPayload(message_t* msg);
}
```

```
  ...
  if (settings.alert & ALERT_RADIO)
    call TheftSend.send(AM_BROADCAST_ADDR, &theftMsg, 0);
}
event message_t *TheftReceive.receive
(message_t* msg, void *payload, uint8_t len) {
  theftLed();
  return msg;
}
```

AMSend is just like send, but with a destination
The theft message has no data, so no use of the payload functions.
On Receive, we just light the “bright red light”

“Basic” Networking

```
configuration AntiTheftAppC { }
implementation {
  ...
  components new AMSenderC(54) as SendTheft,
    new AMReceiverC(54) as ReceiveTheft;
  AntiTheftC.TheftSend -> SendTheft;
  AntiTheftC.TheftReceive -> ReceiveTheft;
}
```

```
generic configuration AMSenderC(am_id_t id) {
  provides interface AMSend;
  provides interface Packet;
  provides interface AMPacket;
  provides interface PacketAcknowledgements as Acks;
}
...
```

atching are
ic components
eceives.

“Basic” Networking – Buffer Management

Sending:

- Each AMSe
- Each outst
  - up to ap
  - common
    reuse it

Receiving:

- Receive pa
  event mess
- Common p
- Common p

```
message_t buffer;
message_t *lastReceived = &buffer;
event message_t* receive(message_t* msg, ...)
{
  /* Return previous message buffer, save current msg */
  message_t *toReturn = lastReceived;
  lastReceived = msg;

  post processMessage();
  return toReturn;
}

task void processMessage() {
  ... use lastReceived ...
}
```

Networking - Summary

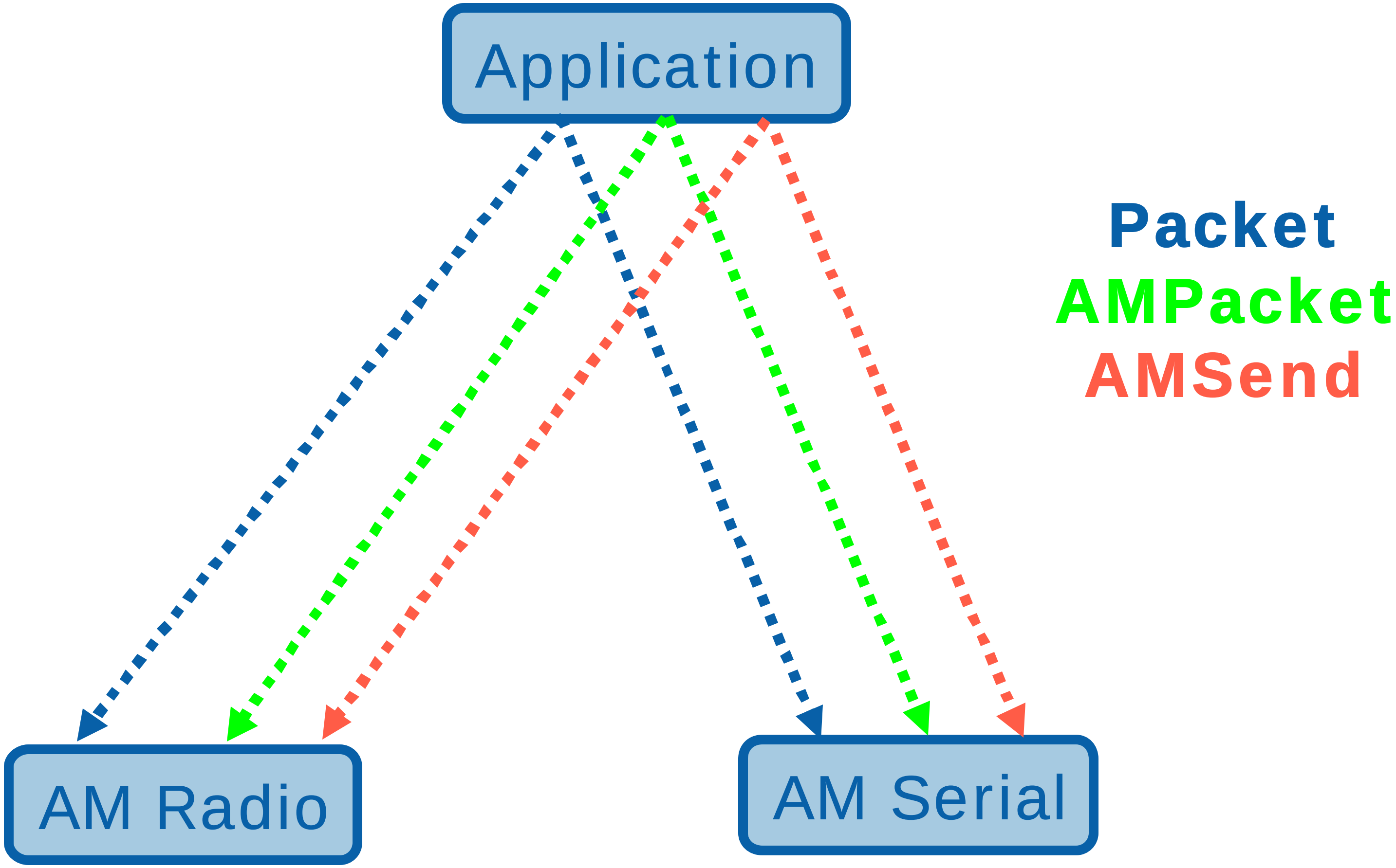
Goals

1. A composable (application-selected) network stack
2. Platform-selected link layers
3. Portable, reusable code above the link layer
4. Cross-platform communication (ex: telosb-micaz, PC-any mote)

Four-part solution:

- abstract data type for packets (message_t)
  - composable, link layer independent, portable
- common networking interfaces (Send, AMSend, Receive)
  - composable, portable
- “external” types (nx_struct, nx_uint16_t, etc)
  - interoperable
- networking component structuring principles
  - composable, link layer independent

Networking Component Structure



Application

AM Radio

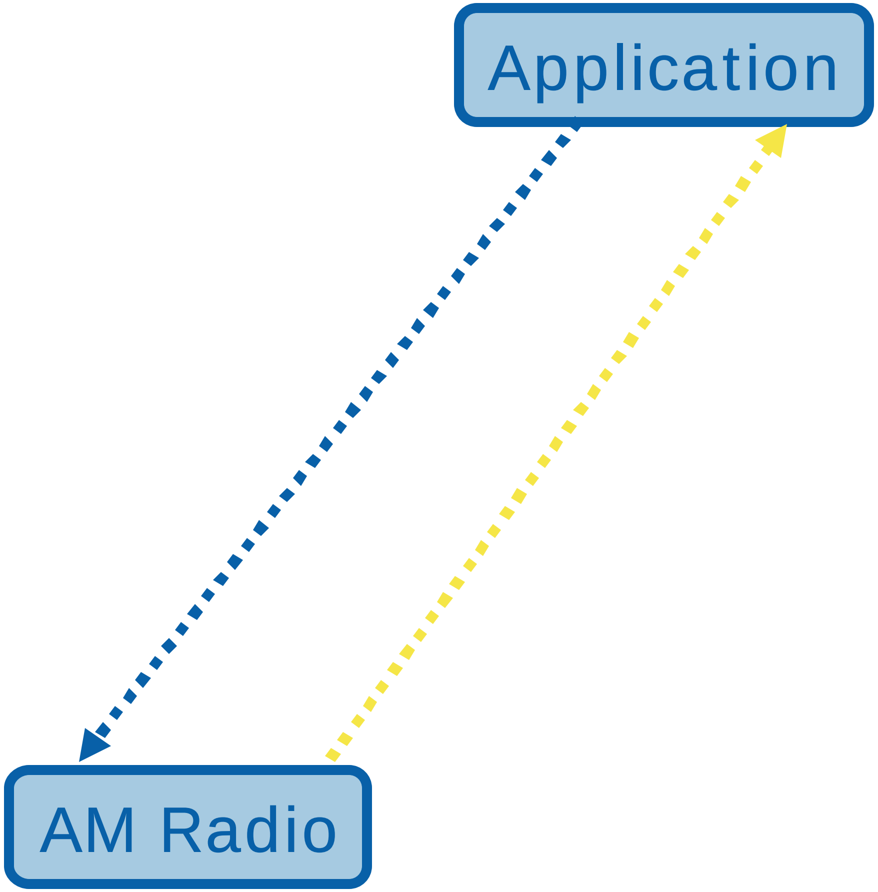
AM Serial

Packet

AMPacket

AMSend

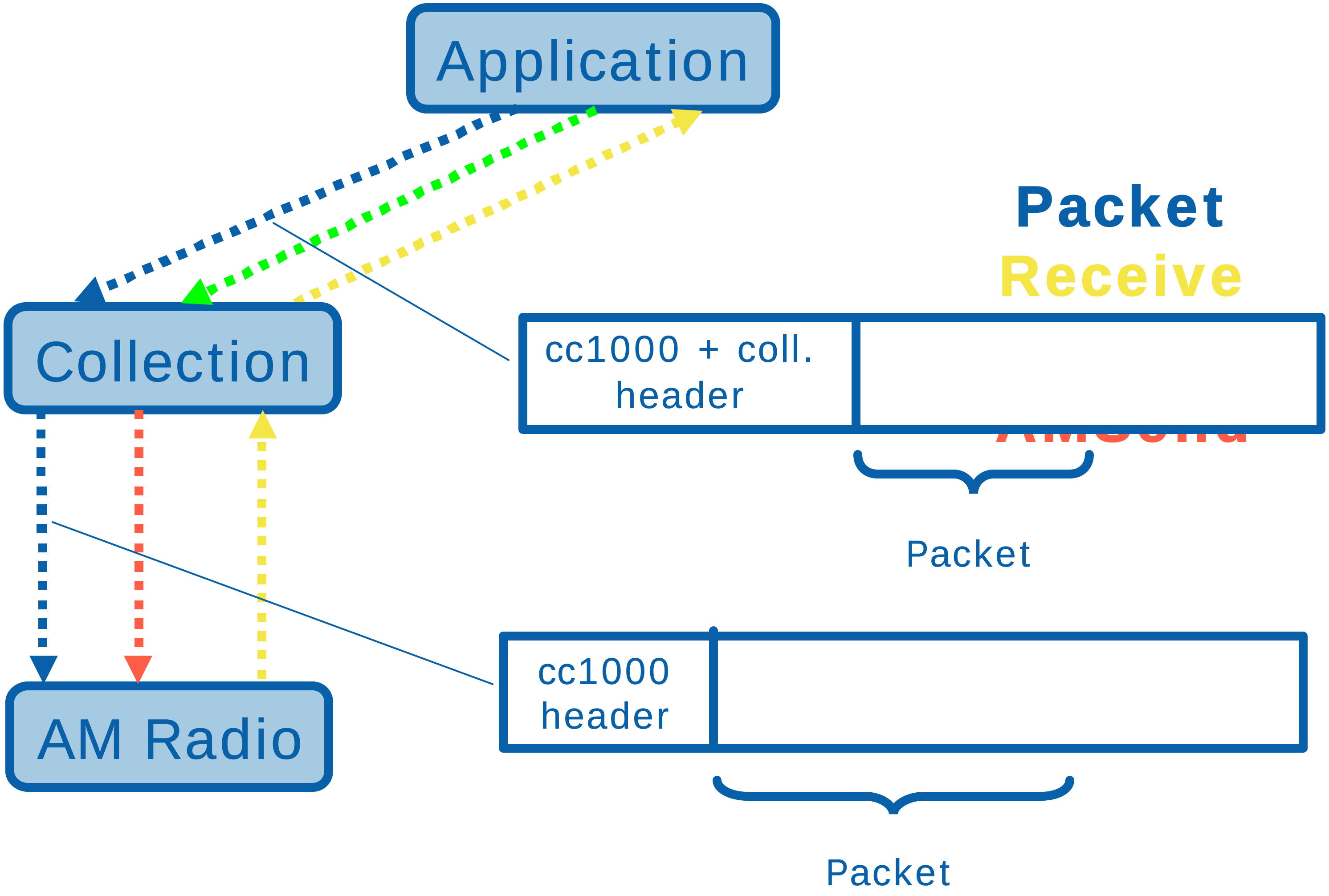
Messaging: networking component structure



Packet

Receive

Messaging: networking component structure



Managing Power

We want our anti-theft device to last a while, or the thief will just wait a bit to steal our motes!

Luckily, in TinyOS 2, this is fairly straightforward

- Services and hardware components switch themselves on and off based on whether they are in use
  - ex: light sensor switched on just before a reading, and off just afterwards
  - ex: accelerometer switched on before group reading, warms up for 17ms, does readings, switches off
- The microcontroller is set to a power mode consistent with the rest of the system
- Radio reception is not as simple, as program doesn't specify when messages might arrive
  - Applications can switch radio on or off explicitly
  - Or, applications can use TinyOS 2's “low-power listening” support
    - Radio channel is checked every N ms
    - Messages sent with an N ms preamble (or repeatedly for N ms)
  - User must specify N (default is N=0, i.e., always on)

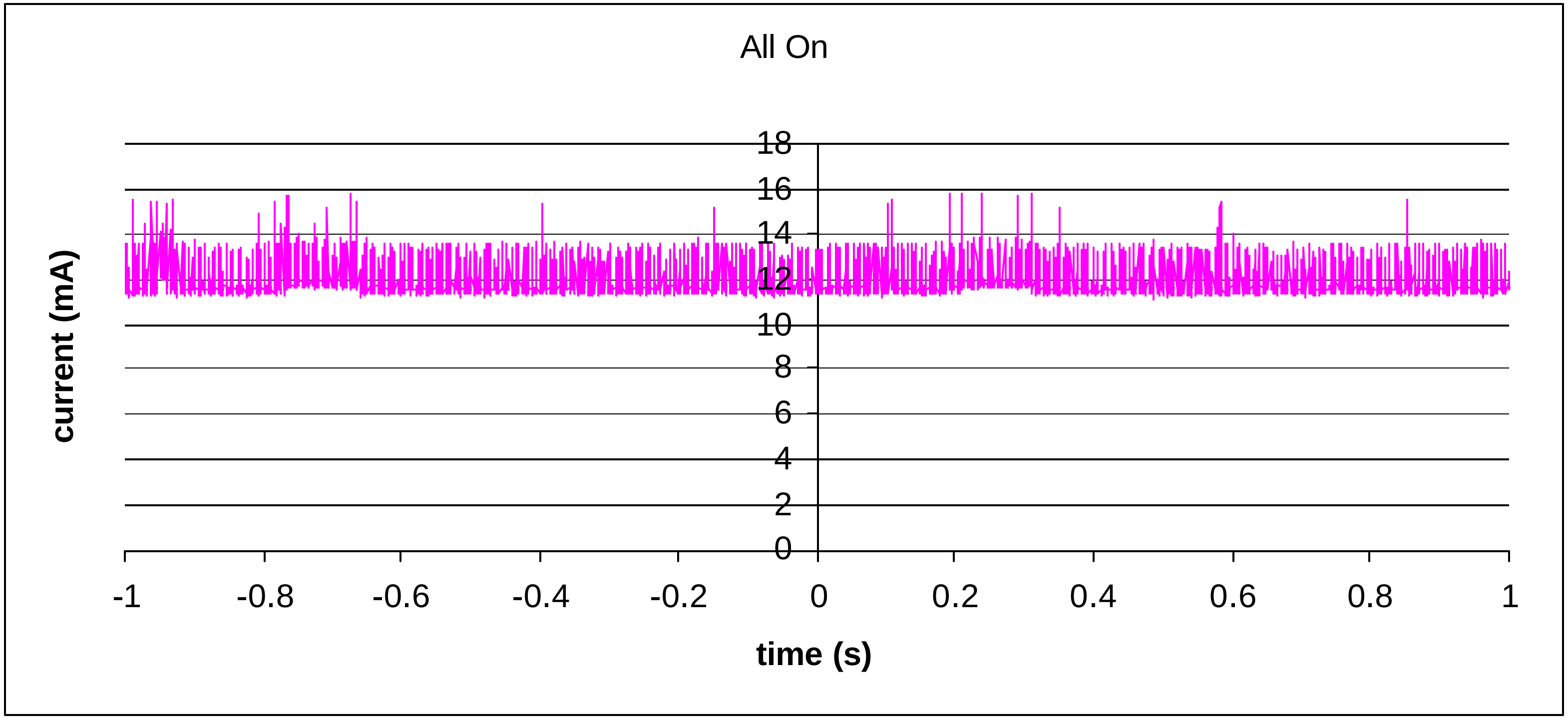
Using Low-power Listening

```
module AntiTheftC ...
  uses interface RadioControl;
  uses interface LowPowerListening;
...
  event void RadioControl.startDone(error_t ok) {
    if (ok == SUCCESS)
      {
        call CollectionControl.start();
        call LowPowerListening.setLocalDutyCycle(200);
      }
  }
```

We request that the radio use a 2% duty-cycle low-power listening strategy

We wire the interface to the actual radio (not shown)

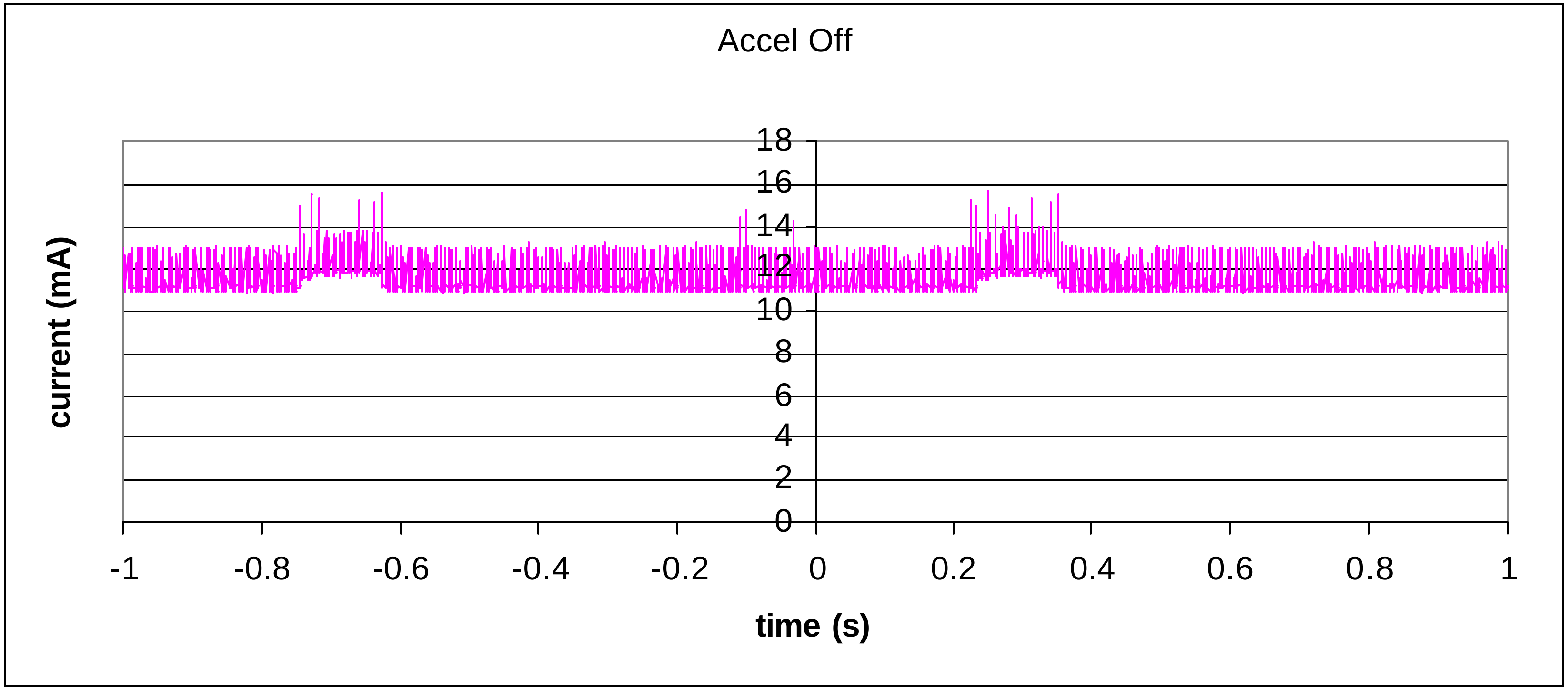
Power Management effects



All power management switched off: 11.83mA

Checking acceleration every second

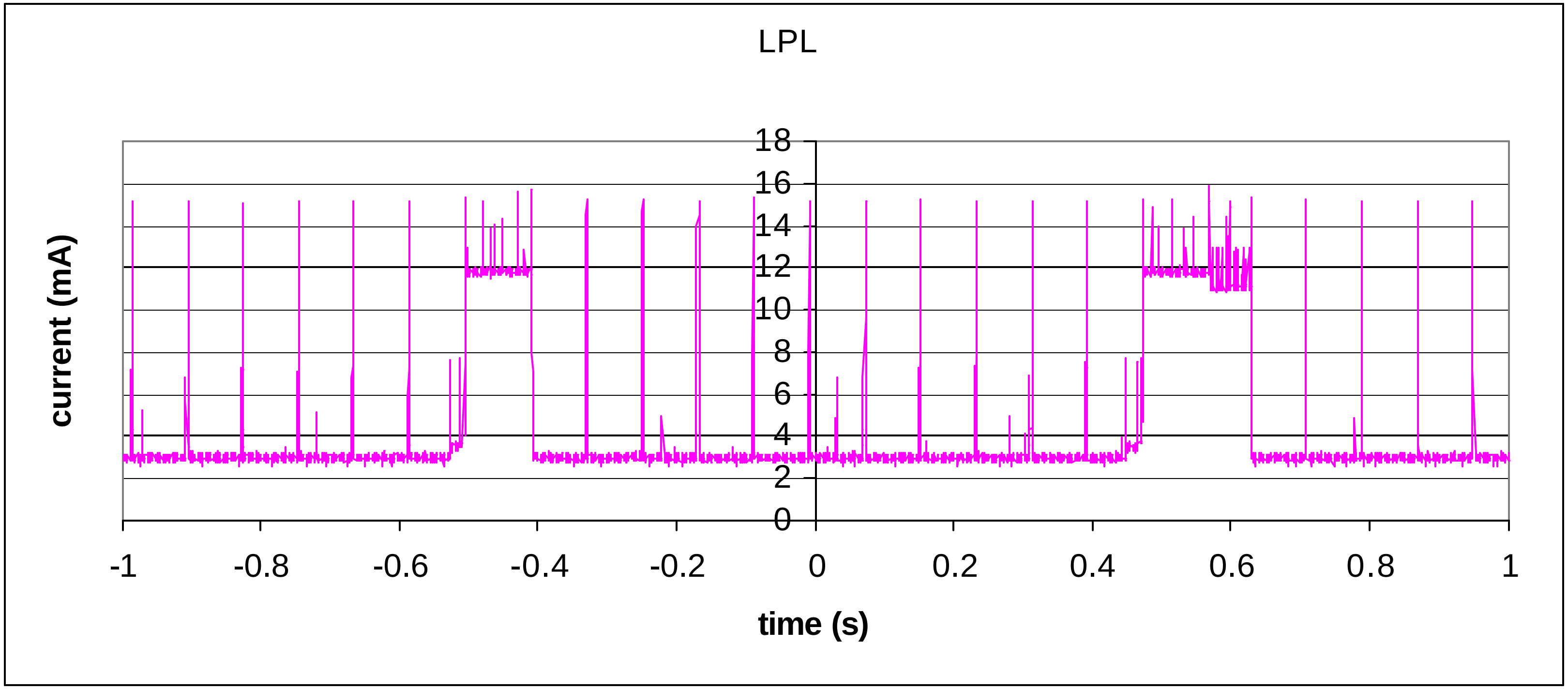
Power Management effects



Accelerometer power management enabled: 11.46mA

Checking acceleration every second

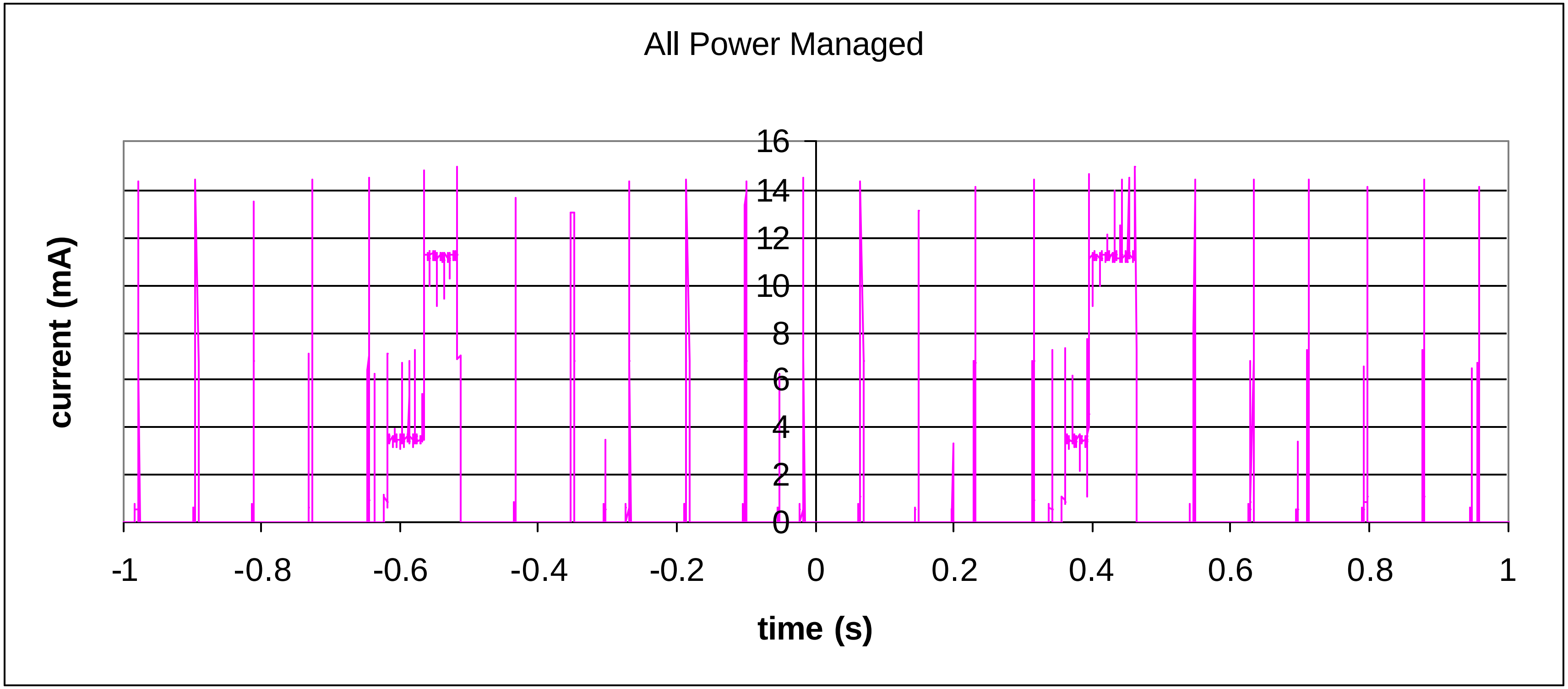
Power Management effects



Low-power listening enabled: 4.26mA

Checking acceleration every second

Power Management effects



Processor power management enabled,

all power management switched on: 1.04mA

Checking acceleration every second

“For Experts”: implementing device drivers

Implementing a service, such as the timer or the light sensor involves one or more of:

- setting up support for multiple clients (via generic components)
- managing concurrent requests to the service (resource management)
- powering any necessary hardware on and off (power management)
- accessing low-level hardware, dealing with interrupts (concurrency)

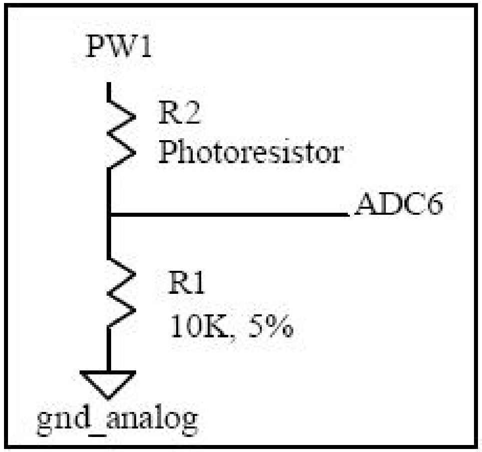
To see these issues in some detail, we’ll look at the example of a simple analog sensor connected to an A/D channel of the microcontroller

- similar, but simpler than the PhotoC sensor used in AntiTheftC (that sensor shares and A/D channel with a temperature sensor, and needs a warmup period, complicating resource and power management)

A simple light sensor

Basic steps to sample light sensor:

- Setup voltage on PW1 pin
- Turn on A/D converter
- Configure A/D converter for channel 6
- Initiate A/D sampling
- In A/D interrupt handler:
  - read A/D result registers
  - report reading to application
- Turn off A/D convert
- Turn off voltage on



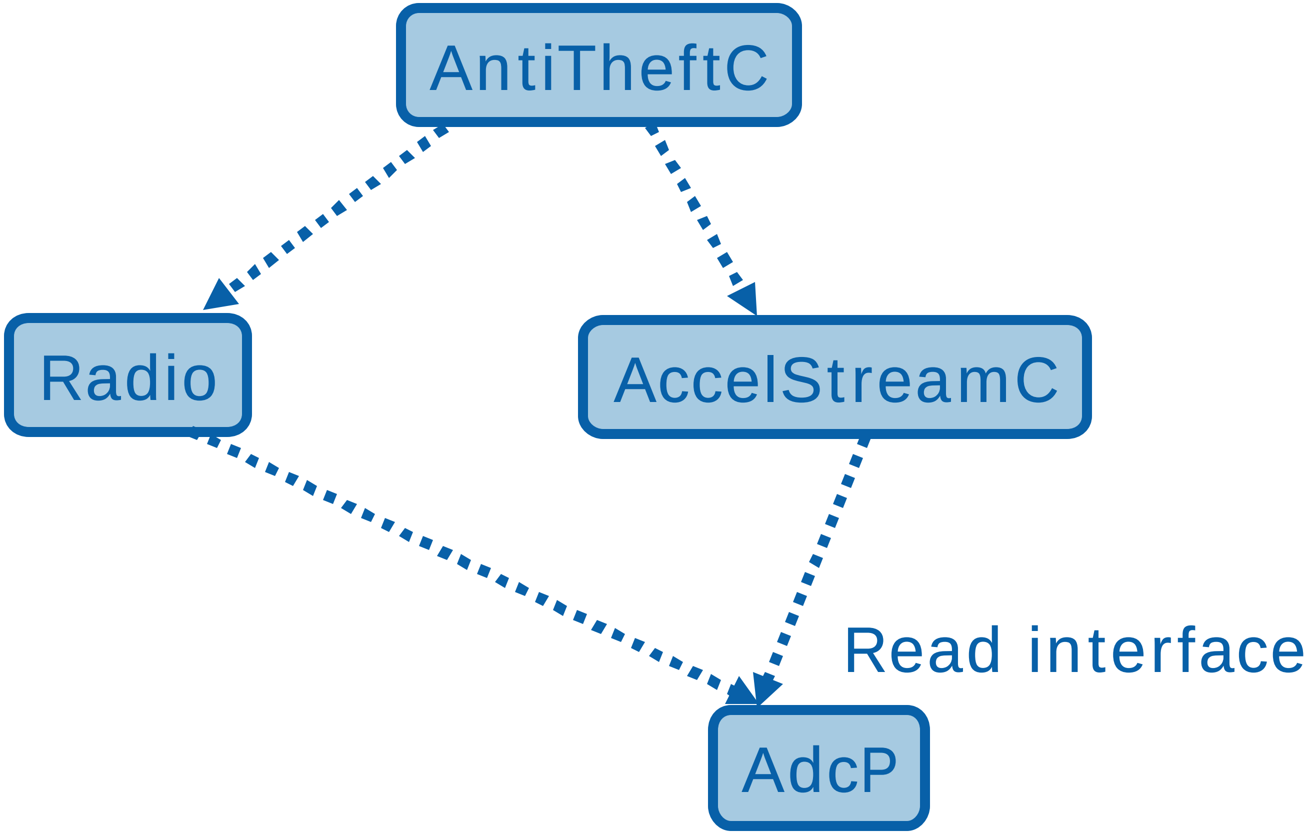
PW1, ADC6 are microcontroller pins

Exposing sensor (maybe) and A/D converter (definitely) to multiple clients
Resource management: need to share light sensor and A/D converter with other clients
Power management: should leave sensor or A/D on between consecutive clients or for repeated accesses
Must avoid data races in interrupt handler and interrupt handler setup code

Services with Multiple Clients

```
module AdcP {
  provides interface Read<uint16_t>;
}
implementation {
  command error_t Read.read() {
    ...
  }
  ...
  task void acquiredData() {
    signal Read.readDone(SUCCESS, val);
  }
}
```

Services with Multiple Clients



AntiTheftC

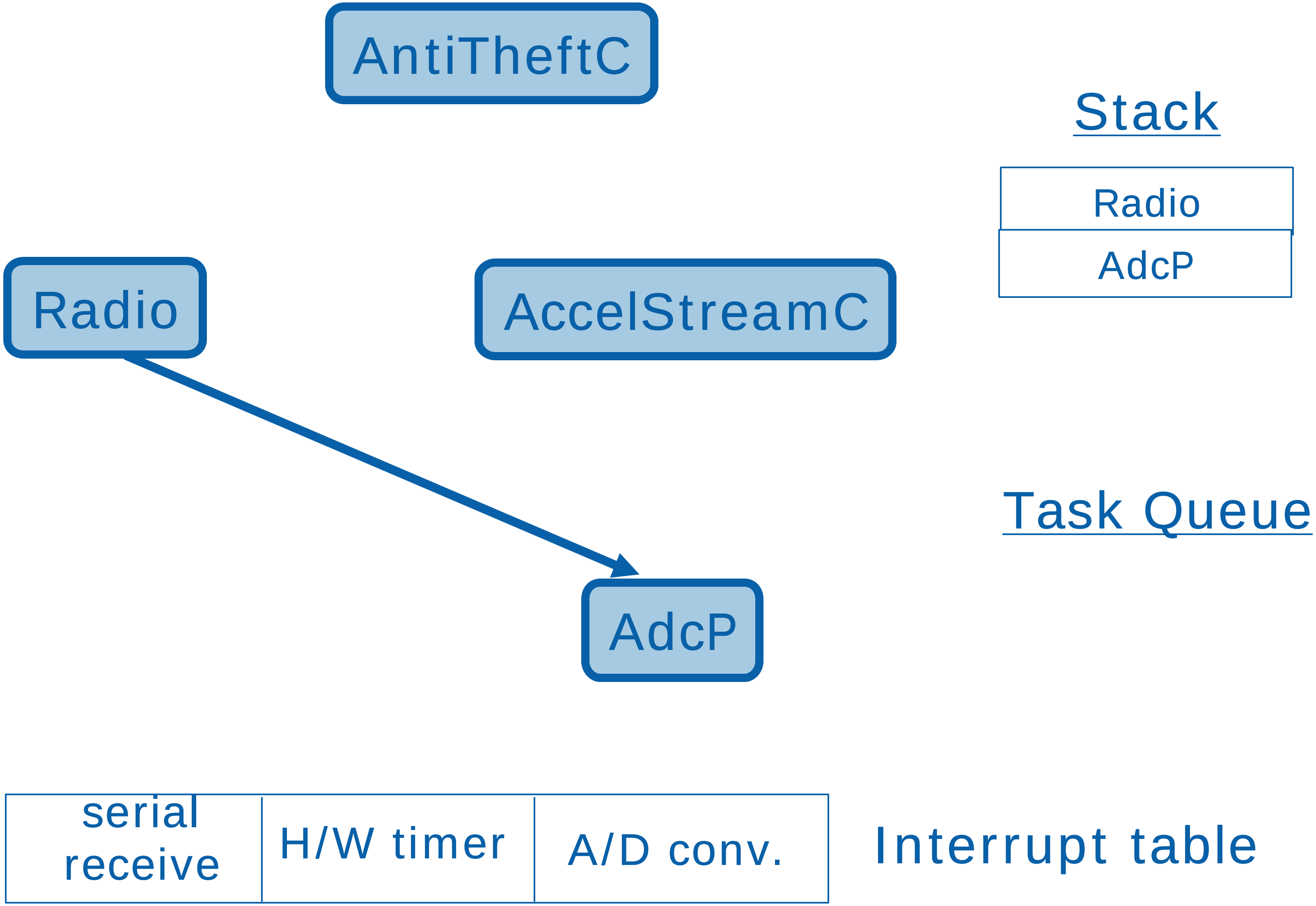
Radio

AccelStreamC

Read interface

AdcP

Services with Multiple Clients



AntiTheftC

Radio

AccelStreamC

AdcP

Stack

Radio

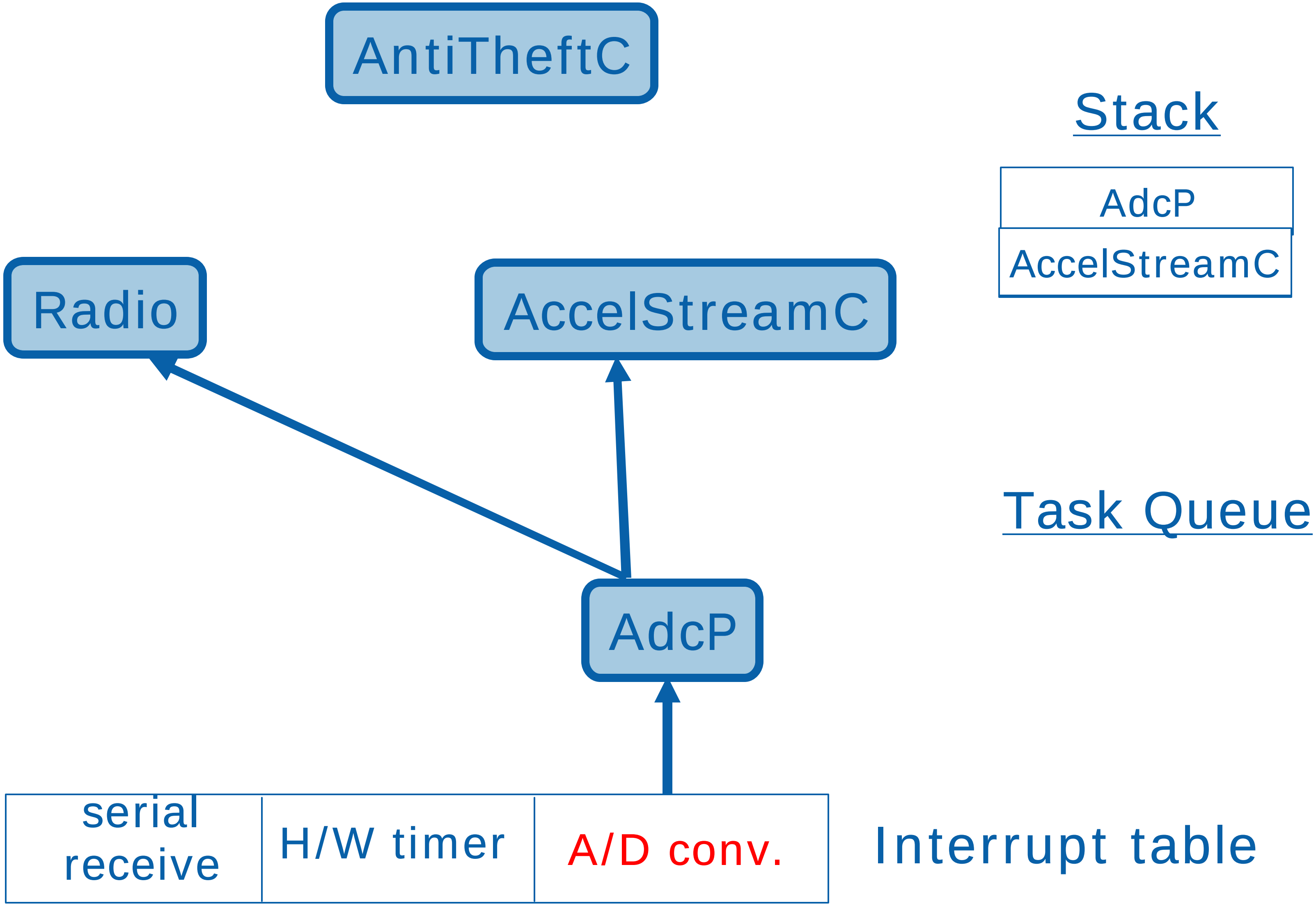
AdcP

Task Queue

serial receive	H/W timer	A/D conv.

Interrupt table

Services with Multiple Clients



AntiTheftC

Radio

AccelStreamC

AdcP

Stack

AdcP

AccelStreamC

Task Queue

serial receive	H/W timer	A/D conv.

Interrupt table

A Fix: Parameterised Interfaces

provide interface Read<uint16_t>[uint8_t id]

- provides an array of interfaces, each identified by an integer
- each of these interfaces can be wired differently
- compiles to a runtime dispatch on the identifier, or an extra argument on function calls

Often, components just want any interface, as long as it’s not used by someone else:

- unique(“some string”): returns a different number at each use with the same string, from a contiguous sequence starting at 0
- uniqueCount(“some string”): returns the number of uses of unique(“some string”)

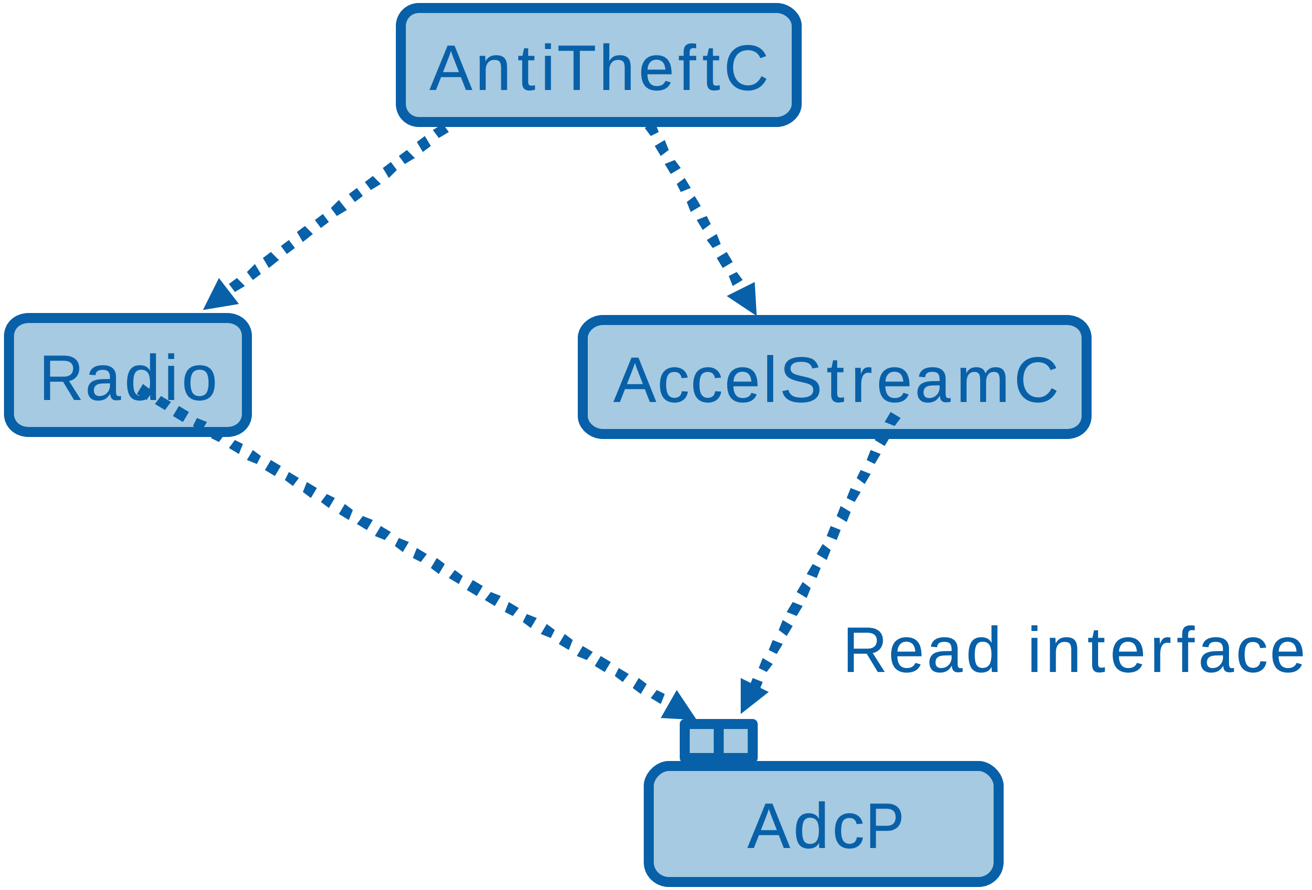
Services with multiple clients

```
module AdcP {
  provides interface Read<uint16_t>[uint8_t client];
}
implementation {
  uint8_t client;
  command error_t Read.read[uint8_t c]() {
    client = c;
    ...
  }
  ...
  task void acquiredData() {
    signal Read.readDone[client](SUCCESS, val);
  }
}
```

Services with multiple clients

```
generic configuration AdcReadClientC() {
  provides interface Read<uint16_t>;
}
implementation {
  components AdcP;

  enum {
    ID = unique(“adc.resource”)
  };

  Read = AdcP.Read[ID];
}
```

Services with Multiple Clients



AntiTheftC

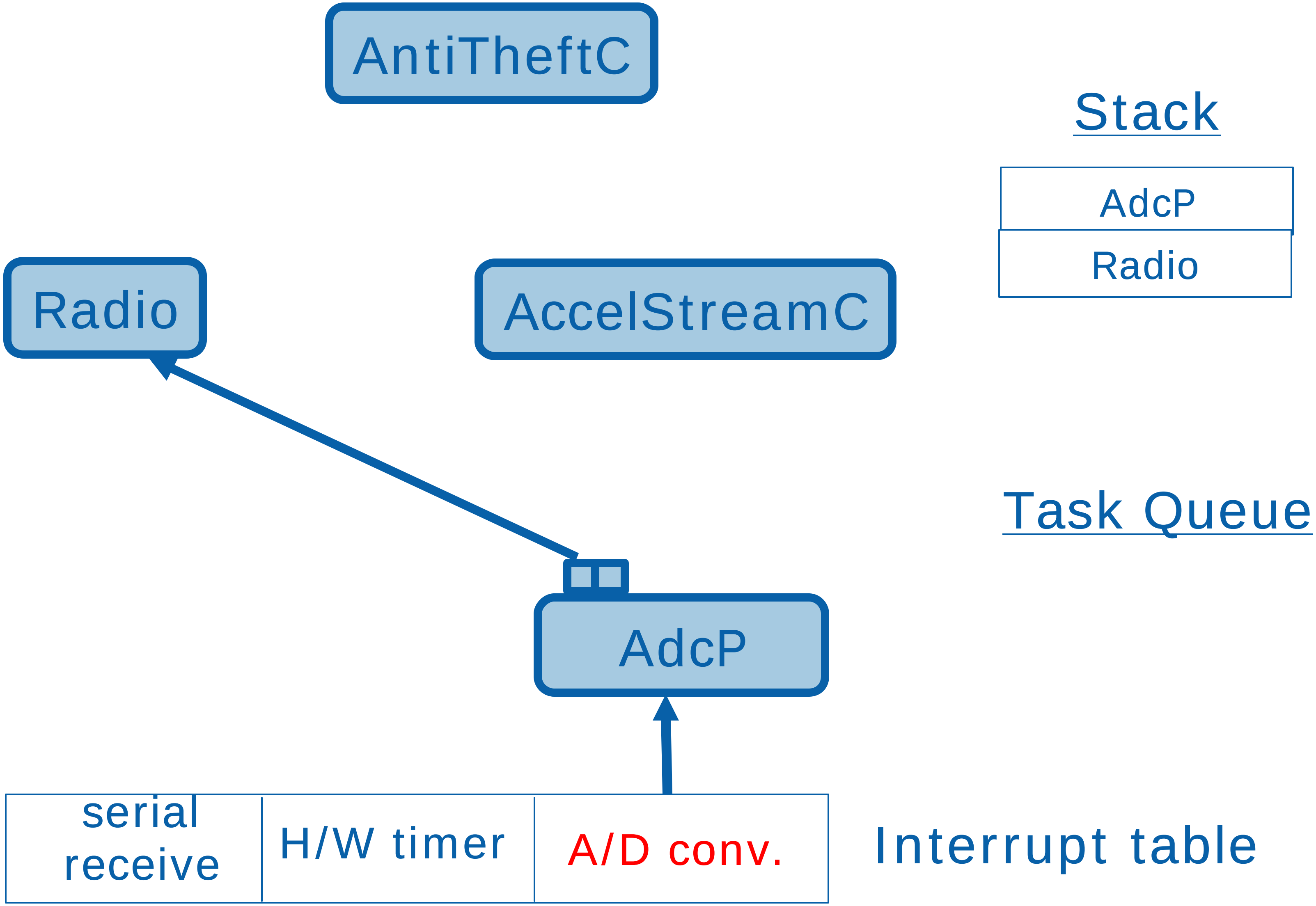
Radio

AccelStreamC

Read interface

AdcP

Services with Multiple Clients



AntiTheftC

Stack

AdcP

Radio

Radio

AccelStreamC

Task Queue

AdcP

serial receive	H/W timer	A/D conv.	Interrupt table

Resource Management

Single application, but still many services competing for resources, e.g.:

- timers in application and multihop routing
- storage in network reprogramming and delay-tolerant networking
- A/D converter used for sensing and CSMA radio

Different requirements from different services:

- exclusive access: CC2420 radio on micaz physically connected to capture pin for hardware timer 1 ⇒ must reserve timer 1 for radio
- latency sensitive: low-jitter multi-kHz A/D sampling
- best effort: wake me every 5 minutes for sampling, and every 12 for route maintenance

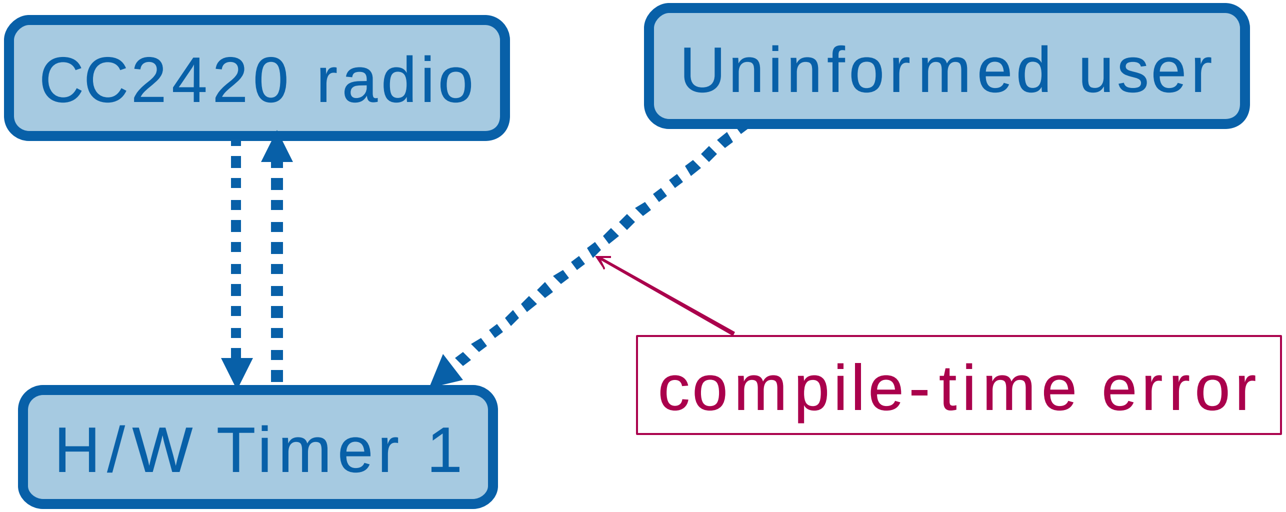
3 kinds of resources:

- *arbitrated, dedicated, virtualised*

Resource Management (continued)

Dedicated Resources

- single client picked at compile-time
- optional compile-time checks



CC2420 radio

Uninformed user

compile-time error

H/W Timer 1

Resource Management (continued)

Dedicated Resources

- single client picked at compile-time
- optional compile-time checks

Properties:

- guaranteed availability
- no latency

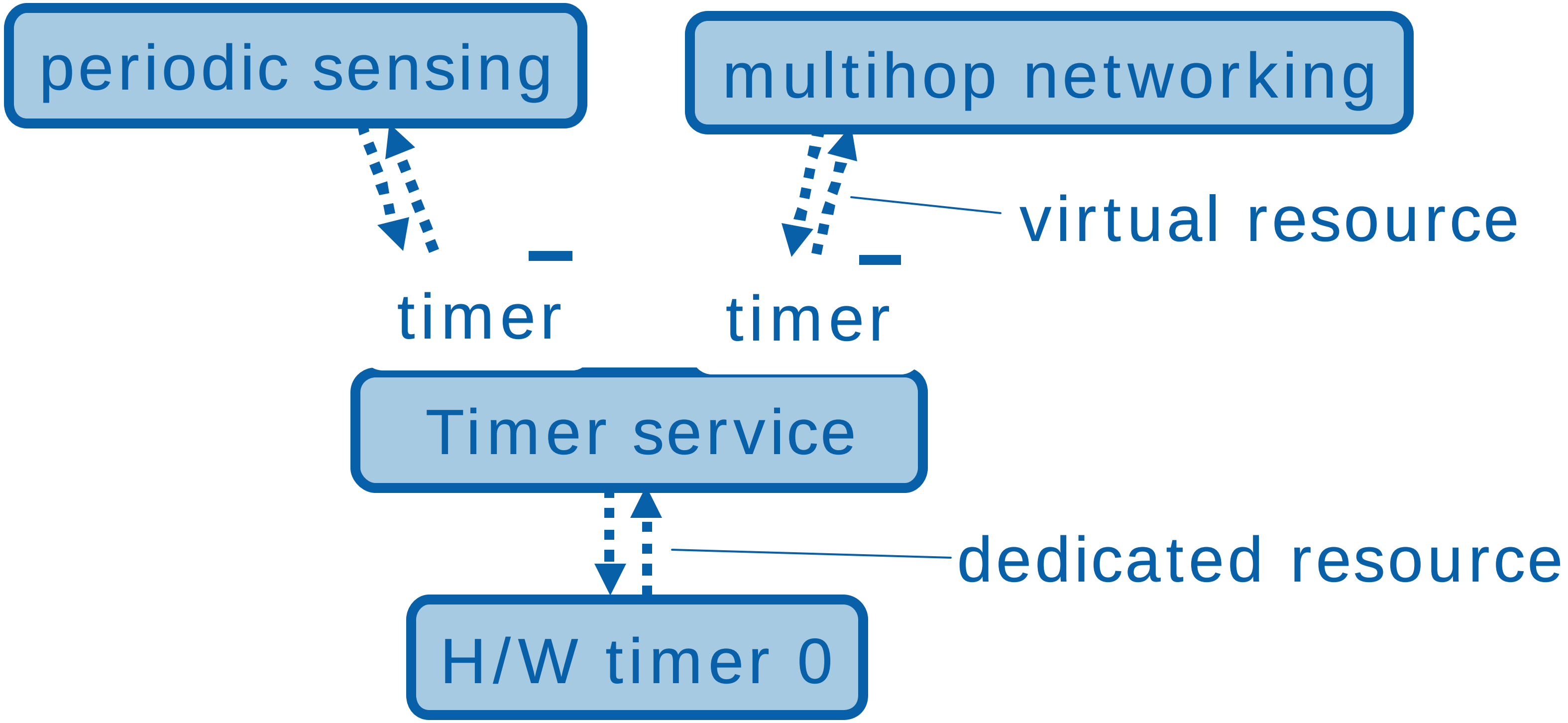
Examples:

- most lowest-level hardware abstractions, e.g., hardware timers

Resource Management (continued)

Virtualised Resources

- service implementation virtualises resource between *N* clients
- all clients known at compile-time



Resource Management (continued)

Virtualised Resources

- service implementation virtualises resource between *N* clients
- all clients known at compile-time

Properties

- guaranteed availability
- sharing-induced latency
- run-time overhead

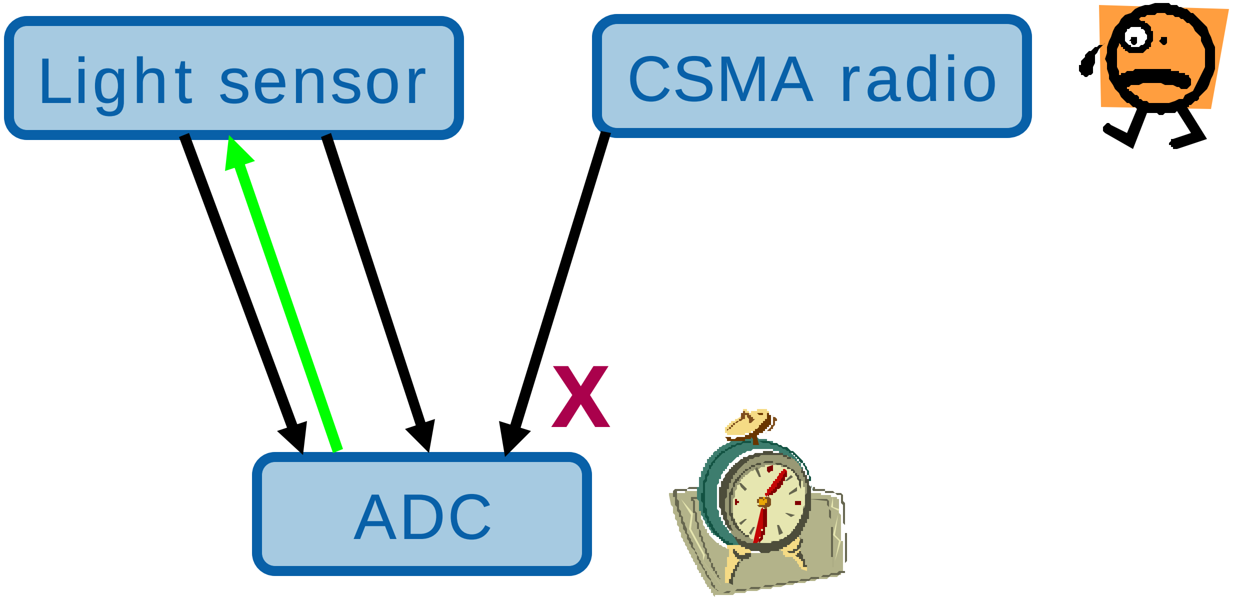
Examples

- scheduler, timers, radio send queue

Resource Management (continued)

Arbitrated Resources

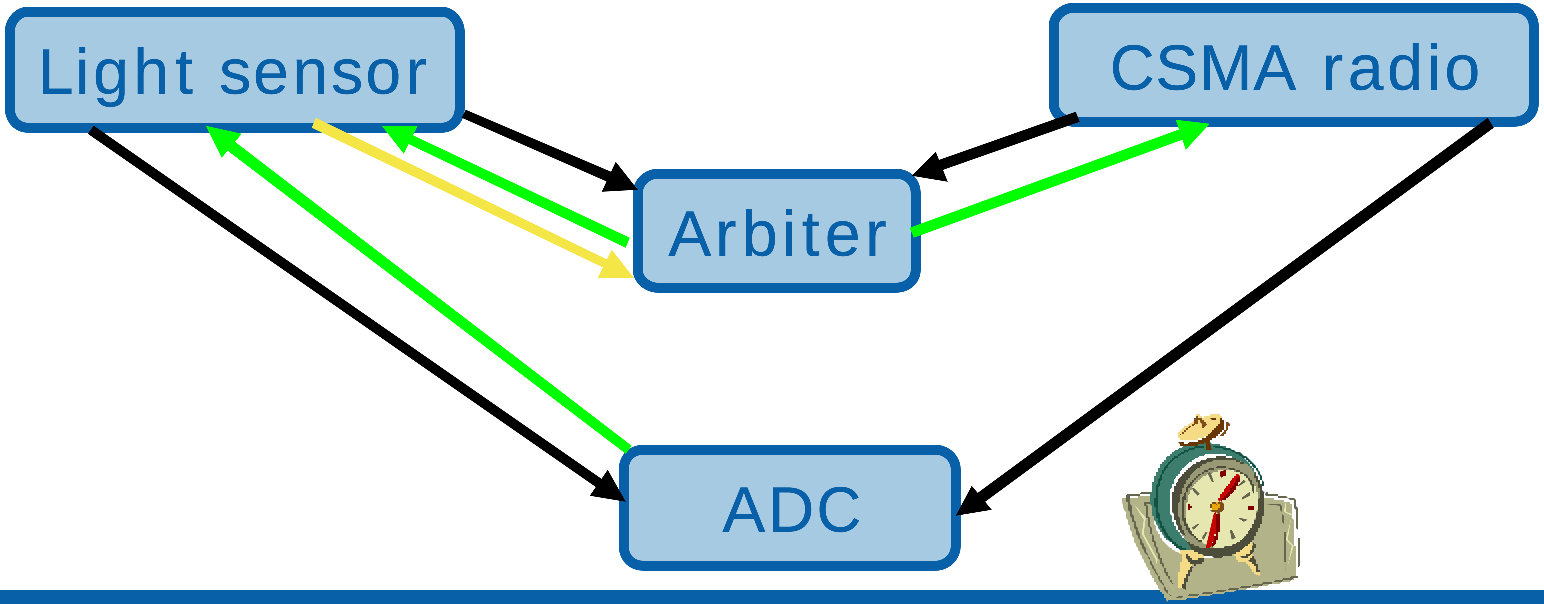
- a shared resource
- some number *N* of clients known at compile-time



Resource Management (continued)

Arbitrated Resources

- a shared resource
- some number *N* of clients known at compile-time (see unique)
- resource *arbiter* manages resource allocation



Light sensor

CSMA radio

Arbiter

ADC

Resource Management (continued)

Arbitrated Resources

- a shared resource
- some number *N* of clients known at compile-time
- resource *arbiter* manages resource allocation

Properties:

- guaranteed availability
- unknown latency
  - immediateRequest: get it **now**, if available

Examples:

- storage, sensing, buses

Resource management for A/D conversion

```
module AdcP {
  provides interface Read<uint16_t>[uint8_t client];
  uses interface Resource[uint8_t client];
}
implementation {
  uint8_t client;
  command error_t Read.read[uint8_t c]() {
    return call Resource.request[c]();
  }
  event void Resource.granted[c]() { client = c; ... }
  ...
  task void acquiredDat
    call Resource.release
    signal Read.readDon
  }
}
```

```
interface Resource {
  async command error_t request();
  async command error_t immediateRequest();
  event void granted();
  async command error_t release();
}
```

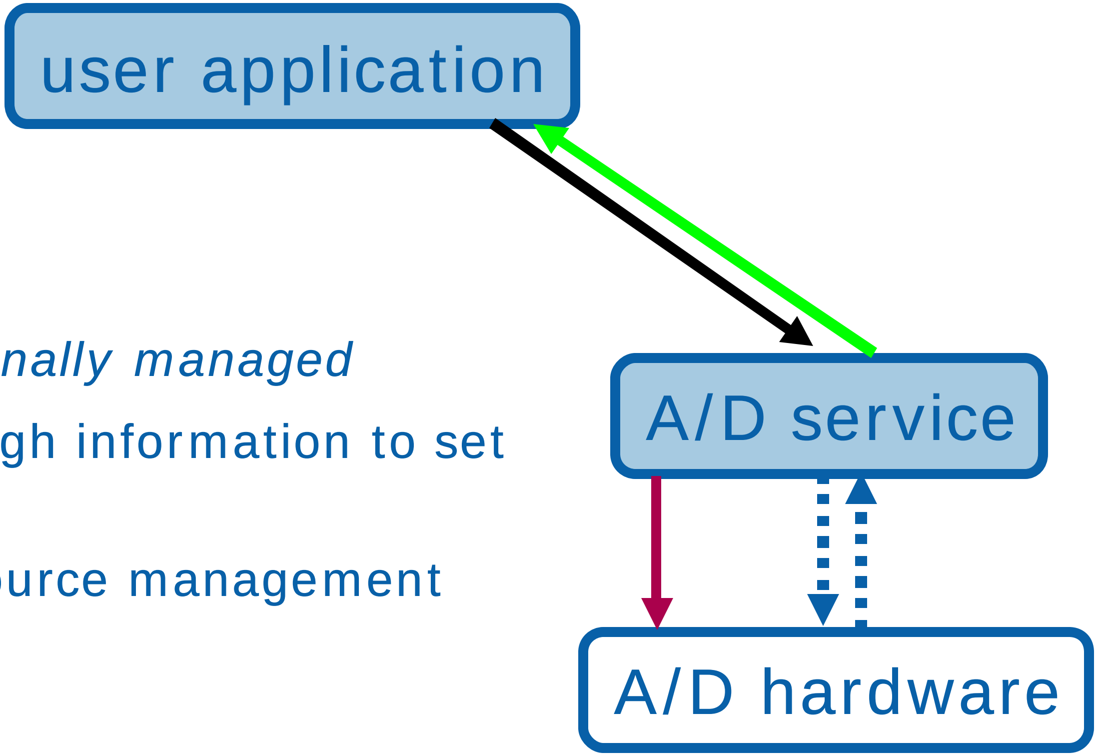
Resource management for A/D conversion

```
configuration AdcC {
  provides interface Read<uint16_t>[uint8_t client];
}
implementation {
  components AdcP,
    new RoundRobinArbiterC(“adc.resource”) as Arbiter;
  Read = AdcP;
  AdcP.Resource -> Arbiter.Resource;
}
```

```
generic configuration RoundRobinArbiterC(char resourceName[]) {
  provides interface Resource[uint8_t client];
  ...
}
implementation
{
    ... uniqueCount(resourceName) ...
}
```

Power Management

Goal: set hardware to lowest-power state consistent with application needs

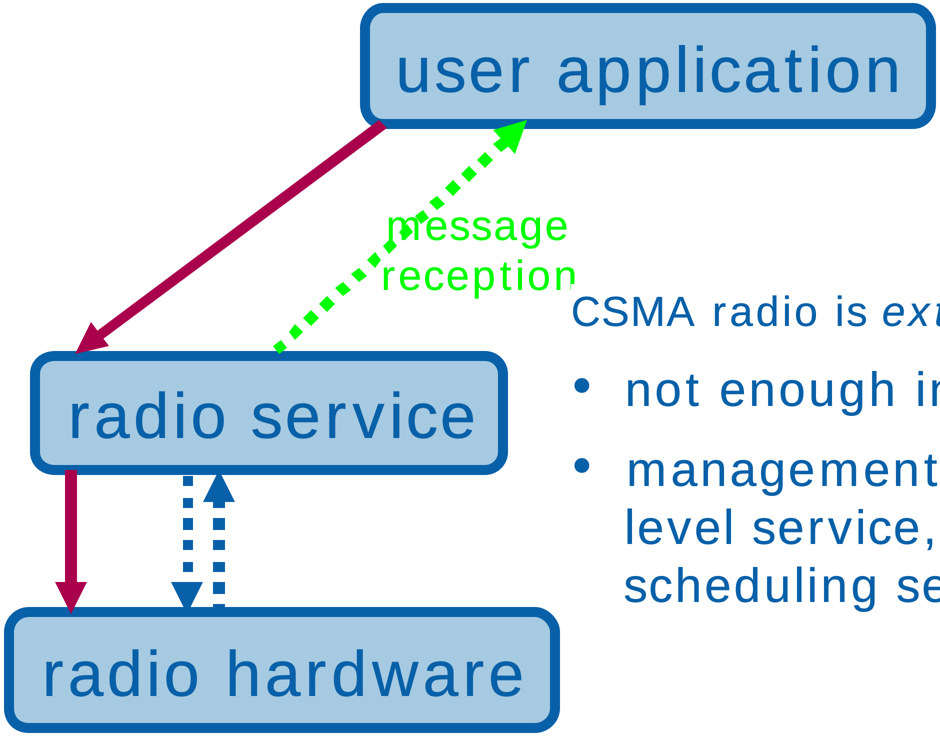


A/D service is *internally managed*

- service has enough information to set hardware state
- can build on resource management system

Power Management

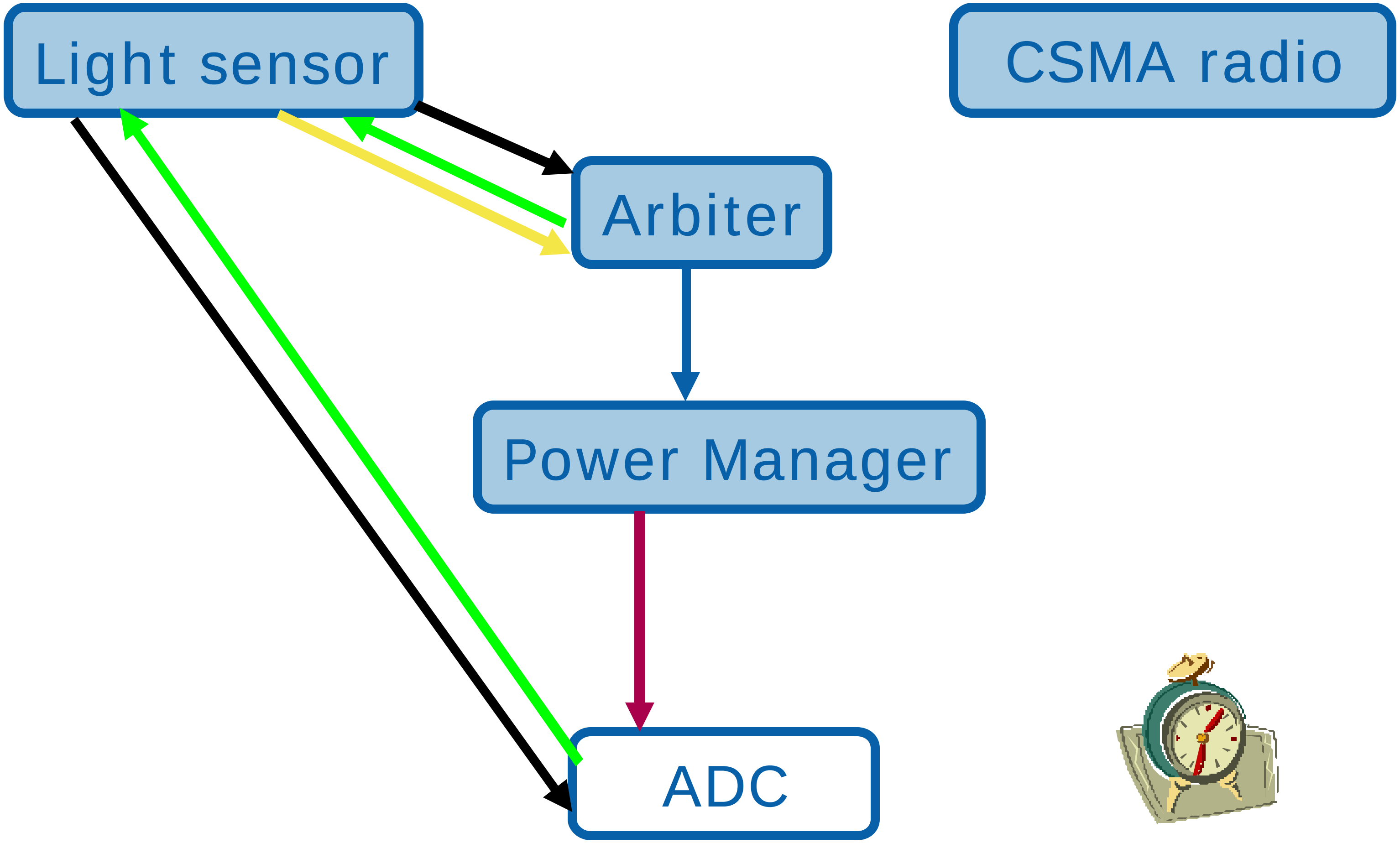
Goal: set hardware to lowest-power state consistent with application needs



CSMA radio is *externally managed*

- not enough information to set hardware state
- management from application, or higher-level service, e.g., neighbourhood message scheduling service (as we saw earlier)

Power Management with Arbiters


Light sensor

CSMA radio

Arbiter

Power Manager

ADC

Resource & Power Management Summary

Three kinds of resources, all have guaranteed availability:

- dedicated
  - single client, no latency
  - typically external power management
- arbitrated
  - multiple clients, unknown latency
  - typically internal power management
    - reusable power managers
- virtualised
  - multiple clients, no latency, runtime overhead
  - typically internal power management

Low-level code and concurrency

Most TinyOS code can live in tasks, and not worry too much about concurrency issues. For instance, in AdcP, the lines

```
call Resource.release[client]();
signal Read.readDone[client](SUCCESS, val);
```

do not need to worry about requests coming in between the release and readDone (and changing client), as:

- tasks do not interrupt each other
- commands and events that are called from interrupt handlers **must** be marked *async*

However, some code has to run in interrupts:

- because it is very timing sensitive
- because the microcontroller signals events via interrupts

nesC support for concurrency

nesC does three things to simplify dealing with interrupt-related concurrency:

- requires the use of *async* on commands and events called from interrupt handlers
- runs a simple data-race detector to identify variables accessed from interrupt handlers
- provides an atomic statement to guarantee the atomic execution of one or more statements

Concurrency example

```
uint8_t resQ[SIZE];

async command error_t Queue.enqueue(uint8_t id) {
  if (!(resQ[id / 8] & (1 << (id % 8))) {   // ← concurrent access!
    resQ[id / 8] |=  1 << (id % 8);        // ← concurrent access!
    return SUCCESS;
  }
  return EBUSY;
}
```

If an interrupt occurs during the if, and the interrupt handler also calls Queue.enqueue then:

- An available slot may be ignored (probably not a problem)
- The same slot may be given twice (oops!)

If an interrupt happens during the 2nd concurrent access (write):

- The interrupt handler’s write of resQ will probably be lost

Data Race Detection

Every concurrent state access is a potential race condition

Concurrent state access:

- If object O is accessed in a function reachable from an interrupt entry point, then all accesses to O are potential race conditions
- All concurrent state accesses must occur in **`atomic`** statements

Concurrent state access detection is straightforward:

- Call graph fully specified by configurations
- Interrupt entry points are known
- Data model is simple (variables only)

Data race fixed

```
uint8_t resQ[SIZE];

async command error_t Queue.enqueue(uint8_t id) {
  atomic {
    if (!(resQ[id / 8] & (1 << (id % 8))) {
      resQ[id / 8] |=  1 << (id % 8);
      return SUCCESS;
    }
  return EBUSY;
}
```

Atomic execution ensured by simply disabling interrupts...

- Long atomic sections can cause problems! E.g.:
  - limit maximum sampling frequency
  - cause lost packets

Concluding Remarks

Reflections on TinyOS

TinyOS status

What we didn't see, and where to find out more

Other sensor network operating systems

Last words

Reflection – Components vs Threads

TinyOS has no thread support

- Execution examples earlier show execution of tasks, interrupt handlers
- This execution crosses component boundaries
- Each component encompasses activities initiated in different places, these could be viewed as independent “threads”. In AntiTheftC we see:
  - booted event initiated in system setup
  - timer event initiated in timer subsystem
  - settings-changed event initiated in dissemination subsystem
  - light and acceleration completion events, ultimately caused by the requests from within AntiTheftC
  - the movement-detection task, initiated in AntiTheftC

However, it’s not always clear exactly what a “thread” of control is. E.g.:

- is the movement-detection task part of the “thread” initiated in response to the periodic timer expiration in the timer subsystem?

Reflection – Components vs Threads

A more productive view is to consider the system as a set of interacting components

- A component maintains the information that represents its state
- A component makes requests for actions from other components
- A component responds to commands and events from other components, representing:
  - Requests (from other components) for the initiation of a new action
    - Ex: please sample the light sensor
  - Completion of requests the component made of other components
    - Ex: message queued for sending to the root of the collection tree
  - Events representing asynchronous actions from the environment or other components
    - Ex: system booted, timer expired, new dissemination value received

Tracking the details of the control flow across components is not necessary within this mindset.

Reflection – Static Allocation

```
module AdcP { … }
implementation {
  uint8_t client;
  uint8_t someState[uniqueCount("adc.resource")];
  .. someState[client] …
}
```

But where did that static allocation happen?

- AntiTheftC allocated some variables for message buffers, acceleration samples
- Instantiation of generic components implicitly allocates state
  - instantiating a module creates a new set of variables
  - unique/uniqueCount allow compile-time sizing of arrays to match the number of clients

Reflection – TinyOS Goals Revisited

Operate with limited resources

- execution model allows single-stack execution

Allow high concurrency

- execution model allows direct reaction to events
- many execution contexts in limited resources

Adapt to hardware evolution

- component, execution model allow hardware / software substitution

Support a wide range of applications

- tailoring OS services to application needs

Be robust

- limited component interactions, static allocation

Support a diverse set of platforms

- OS services should reflect portable services

Reflection – Status

TinyOS 2.0 released in November 2006

- 25 “TinyOS Enhancement Proposals” describing TinyOS structure
- 114k lines of code in TinyOS core (in CVS today)

Services:

- completed: booting, scheduling, timer, A/D conversion, I^{2}C bus, radio, serial port, storage, multihop collection and dissemination, telosb sensors, simple mica sensors
- in progress: over-the-air reprogramming, more sensors
- in limbo: security, time synchronisation

Platforms: mica family, telos, eyes, tinynode, intel mote 2

2.0.1 release planned for IPSN conference

- General API cleanup (based on TEP finalisation), bug fixes

Other OSes for Mote-class Devices

SOS https://projects.nesl.ucla.edu/public/sos-2x/

- C-based, with loadable modules and dynamic memory allocation
- also event-driven

Contiki http://www.sics.se/contiki

- C-based, with lightweight TCP/IP implementations
- optional preemptive threading

Mantis http://mantis.cs.colorado.edu

- C-based, with conventional thread-based programming model
- semaphores+IPC for inter-thread communication

What we didn’t see, Where to find out more

We didn’t see:

- The build system
- How to get code onto motes
- The 3-level hardware abstraction architecture
- Storage (flash) abstractions

Where to find out more

- http://www.tinyos.net
- The TinyOS Enhancement Proposals (TEPs)
- The web tutorials
- Phil Levis’s nesC/TinyOS programming manual, available from http://csl.stanford.edu/~pal/

Last Words

Work In Progress

- sensorboards, TEPs
- community feedback on design and TEPs

Remains to be done:

- finish system services, in particular network reprogramming

But, compared to TinyOS 1.1, TinyOS 2.0 is already:

- better designed
- better documented
- more reliable
- more portable

Download it today: http://www.tinyos.net/dist-2.0.0 !

intel®

Leap ahead™