T2

What the Second Generation Holds

Philip Levis, David Gay, David Culler, Vlado Handziski, Jan Hauer, Cory Sharp, Ben Greenstein, Jonathan Hui, Joe Polastre, Robert Szewczyk, Kevin Klues, Gilman Tolle, Lama Nachman, Adam Wolisz, and a few more...

In the Beginning

(1999)

- Sensor networks are on the horizon...
- ... but what are they going to do?
  - What problems will be important?
  - What will communication look like?
  - What will hardware platforms look like?
- Having an operating system is nice...
- ... but how do you design one with these uncertainties?

The TinyOS Goals

(ASPLOS 2000)

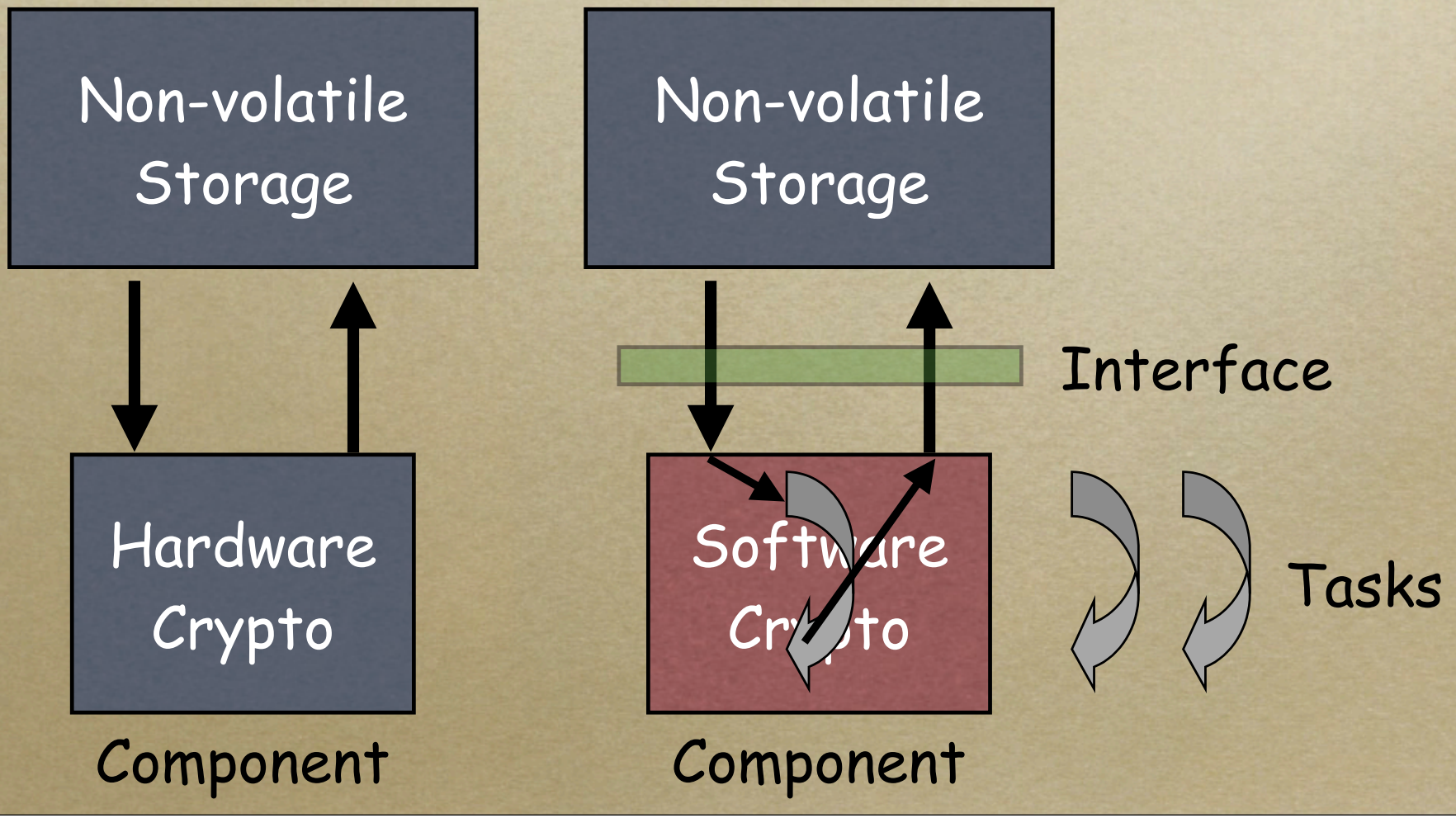
- Allow high concurrency
- Operate with limited resources
- Adapt to hardware evolution
- Support a wide range of applications
- Be robust
- Support a diverse set of platforms

TinyOS Basics

- A program is a set of components
  - Components can be easily developed and reused
    - Adaptable to many application domains
  - Components can be easily replaced
  - Components can be hardware or software
    - Allows boundaries to change unknown to programmer
- Hardware has internal concurrency
  - Software needs to be able to have it as well
- Hardware is non-blocking
  - Software needs to be so as well

TinyOS Basics, Continued

(2002)



The TinyOS Goals

(ASPLOS 2000)

- Allow high concurrency
- Operate with limited resources
- Adapt to hardware evolution
- Support a wide range of applications
- Be robust
- Support a diverse set of platforms

The TinyOS Goals

(The David Gay Scorecard, 2005)

- Allow high concurrency (A)
- Operate with limited resources (A-)
- Adapt to hardware evolution (B)
- Support a wide range of applications (B)
- Be robust (C)
- Support a diverse set of platforms (D)

The T2 Agenda

- Allow high concurrency (A)
- Operate with limited resources (A–)
- Adapt to hardware evolution (B)
- Support a wide range of applications (B)
- Be robust (C)
- Support a diverse set of platforms (D)

Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core structure
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

The Issues Ahead

- Support a wide range of applications (B)
  - Building small applications is easy
  - Building large applications is possible, but hard
- Be robust (C, but a D in Phil's world)
  - New hardware has punched holes in assumptions
  - Many unforeseen interactions
- Support a diverse set of platforms (D)
  - Current trend is to have 1-2 supported platforms
  - More platforms leads to lots of redundant code

Diverse Platforms

- Problem: many new platforms emerging, but few run TinyOS
  - E.g., “How do I port TinyOS to my new platform with an XXX?”
  - E.g.,“Porting TinyOS requires a lot of work!”
- Cause: platforms are self-contained units, with dependencies and assumptions

Future Platforms

- A platform is usually a collection of interconnected hardware chips

Platform	MCU	Radio
mica2	ATMega128	CC1000
micaZ	ATMega128	CC2420
Telos	MSP430	CC2420
eyes	MSP430	Infineon

- A next generation OS needs to separate the code for hardware resources from the code that composes them into a platform.

Robustness

- Problem: failures of structure
  - Components written to operate independently, but manifest conflicts when combined
  - E.g., “The ADC hangs when I send packets!”
- Problem: failures of abstraction
  - Dynamic allocation leads to dynamic failures
  - Generally, TinyOS follows a static allocation policy, except for one place: the task scheduler
- Cause: TinyOS didn’t get everything right the first time

Improving Robustness

- A next generation OS needs to define a structure and architecture for how components interact
  - Mechanisms and division of responsibilities
- A next generation OS needs to identify and redefine problematic abstractions
  - There's a five years of experience from thousands of users

Application Support

- Problem: building large applications out of many independent components is hard
  - E.g., “Why is it that if I turn off the radio to save power, flash storage stops working?”
  - E.g., “Why is it that if I put the send queue above this component it hangs?”
- Problem: inefficiency of defensive code
  - E.g., the radio stack is initialized 6 times
- Cause: components are independent, applications are not

Supporting Applications

- Complexities stem from hidden dependencies over shared resources
  - E.g., SPI bus
- Inefficiencies stem from tension between local independence and global properties
  - All 6 use the radio, but how can the app know?
- A next generation OS needs to have true APIs that encapsulate underlying services in a usable whole

An Example: CC2420

- Platform diversity: needs to be easily portable to new platforms (micaZ + Telos)
- Robustness: 1.x implementation breaks TinyOS concurrency model due to shared task queue
- Applications: interactions with other components are a big source of problems

Three Approaches

- Multi-layer abstractions
  - Present a true spectrum of abstraction layers, from the most general and portable to the most efficient and platform-specific
- Static binding and allocation
  - Anything and everything that can be determined statically, should be
- Service distributions
  - A coherent collection of APIs above the OS core, which implement policies for an intended set of application domains

Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

T2 Core

- Platforms
- Concurrency model
- Scheduler

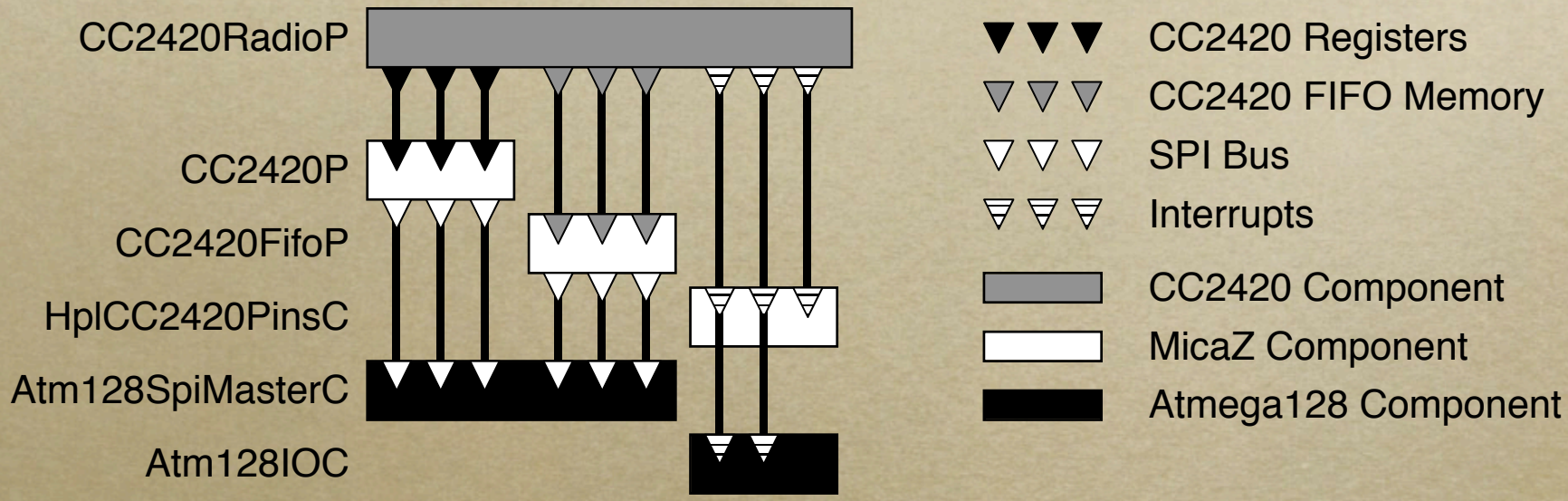
T2 Platforms

- A platform is a collection of chips

Platform	MCU	Radio
mica2	ATMega128	CC1000
micaZ	ATMega128	CC2420
Telos	MSP430	CC2420
eyes	MSP430	Infineon

- Chip implementations are platform independent
- Platforms provide adapter code

Example: CC2420

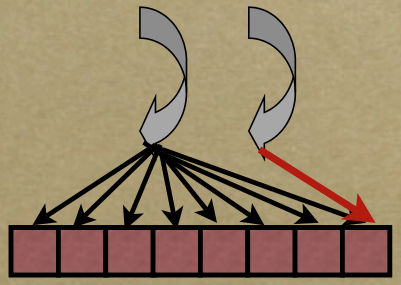


TinyOS 1.x Concurrency

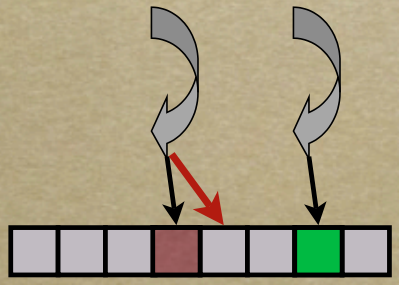
- Tasks run to completion (do not preempt)
  - while(1) {runNextTaskOrSleep();}
- Two kinds of function
  - Synchronous: can only run in a task (main)
  - Asynchronous: can run in a task or interrupt
    - Asynchronous code can preempt
    - Compile-time race condition detection
- Posting is how you go from async to sync

Concurrency Model

- T2 has the same basic concurrency model
  - Tasks, sync vs. async
- T2 changes the task semantics
  - TinyOS 1.x: post() can return FAIL, can post() multiple times (shared slots)
  - T2: post returns FAIL iff the task is already in the queue (single reserved slot per task)



TinyOS 1.x



T2

Scheduler

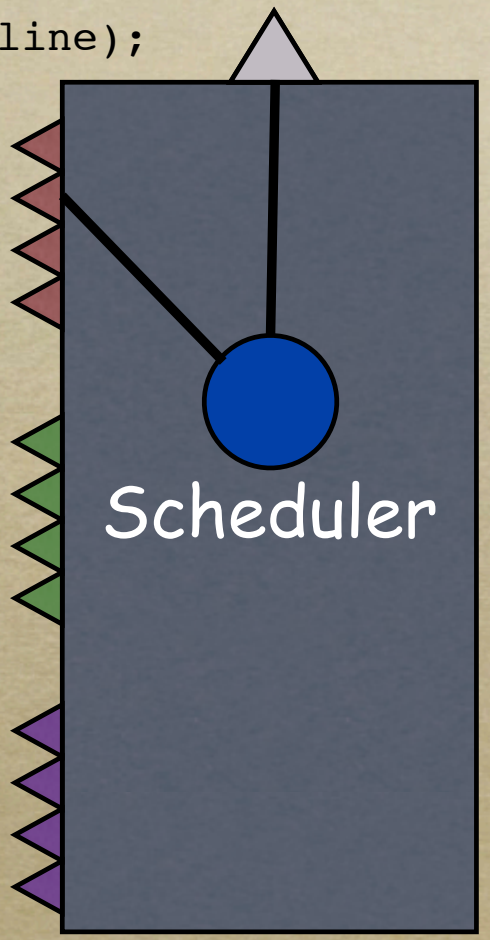
- Common (but mostly disappeared) TinyOS complaint: why no task priorities?
  - Scheduler was the one part you could not change (built into the language)
  - Some application domains may really need them
- T2 solution: scheduler is a component, tasks are just another interface
  - nesC compiler transforms task/post into used interfaces

Scheduler, Continued

```
interface TaskEDF {
  command async error_t post(uint16_t deadline);
  event void run();
}
```

```
interface TaskBasic {
  command async error_t post();
  event void run();
}
```

```
interface TaskPriority {
  command async error_t post();
  event void run();
}
```



Core Effects

- Platform/chip decomposition makes a platform diversity much easier
- Scheduler can enable a wider range of applications

Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

Varying Specificity

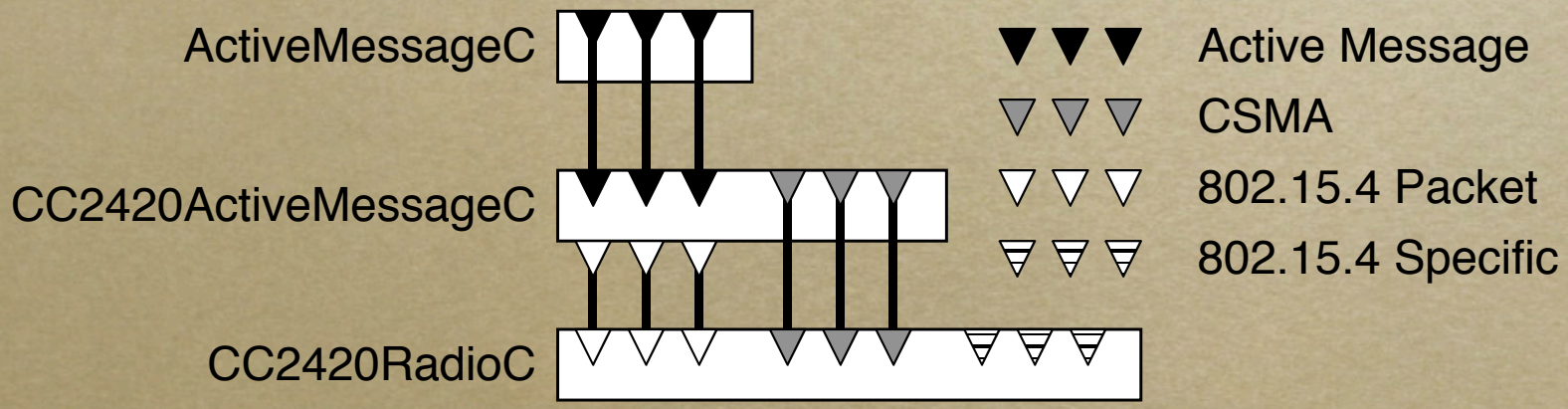
- I send packets
- I send packets and expect acks
- I sometimes turn off CSMA
- I sometimes turn off address decoding

More Varying Specificity

- I need a timer, roughly ms granularity, some jitter is OK
- I need a timer, 32kHz granularity, as low jitter as possible
- I need a timer, 32kHz granularity, that works in low power mode X

Multi-Layer Abstractions

- Many fine-grained layers of increasing power and specificity



Partial Virtualization

- Some abstractions are shared and virtual
  - Basic timers, packet transmission
- Some are dedicated and physical
  - Compare register B2 for CC2420 MAC timing
- Some are shared and physical
  - SPI bus between CC2420 and flash storage

Their Effect

- Make it clear when your code is portable vs. platform specific
  - Improve robustness by making exchanging hidden dependencies for explicit ones
- Enable a wider range of applications
  - Support full spectrum of simple and portable to high-performance subsystems

Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

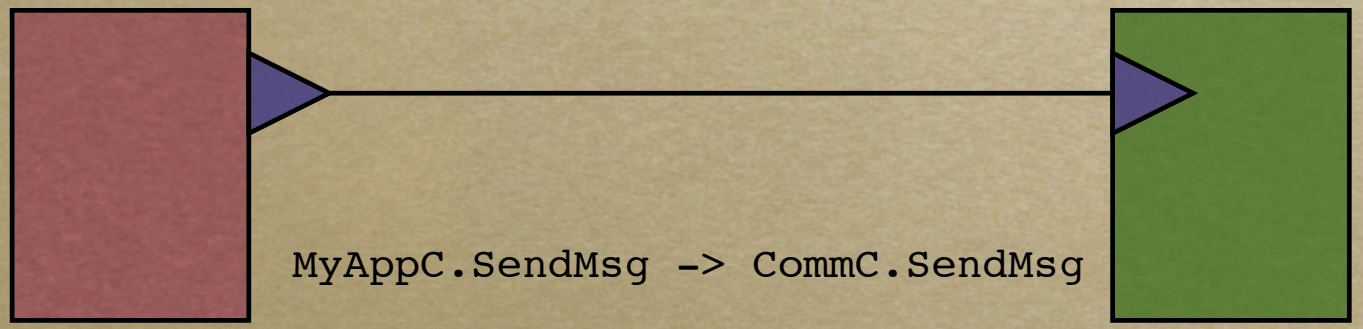
Static Binding

- nesC components can only interact through interfaces

```
interface SendMsg {...}
```

```
configuration MyAppC {
  uses interface SendMsg;
}
```

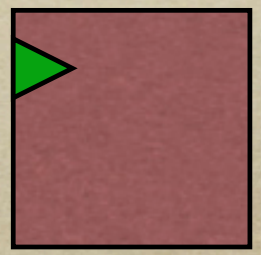
```
configuration CommC {
  provides interface SendMsg;
}
```



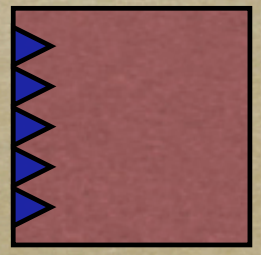
Static Binding, Continued

- Run-time vs. compile time parameters

```
interface CC2420Register {
  command uint16_t read(uint8_t reg);
  command uint8_t write(uint8_t reg, uint16_t val);
  command uint8_t strobe();
}
component CC2420C {
  provides interface CC2420Register;
}
```



```
interface CC2420StrobeReg {
  command uint8_t strobe();
}
component CC2420C {
  provides interface CC2420StrobeReg as SNOP;
  provides interface CC2420StrobeReg as STXONCCA;
   ....
}
```



Static Allocation

- You know what you'll need: allocate it at compile-time (statically)
- Depending on probabilities is a bet
  - I.e., "it's very unlikely they'll all need to post tasks at once" = "they will"
- You know what components will use a resource, can allocate accordingly
  - In some cases, static allocation can **save** memory

Saving Memory

```
module Foo {
  bool busy;

  command result_t request() {
    if (!busy() &&
        post fooTask() == SUCCESS) {
      busy = TRUE;
      return SUCCESS;
    }
    else {
      return FAIL;
    }
  }
```

TinyOS 1.x

```
module Foo {
  bool busy;

  command result_t request() {
    return post fooTask();
  }
```

T2

Effects

- Static binding improves robustness
  - Push as many checks to compile-time as possible
  - Bad parameters become impossible compositions
- Static allocation improves robustness
  - You can make safe assumptions in code
  - Code is shorter, simpler, and more deterministic

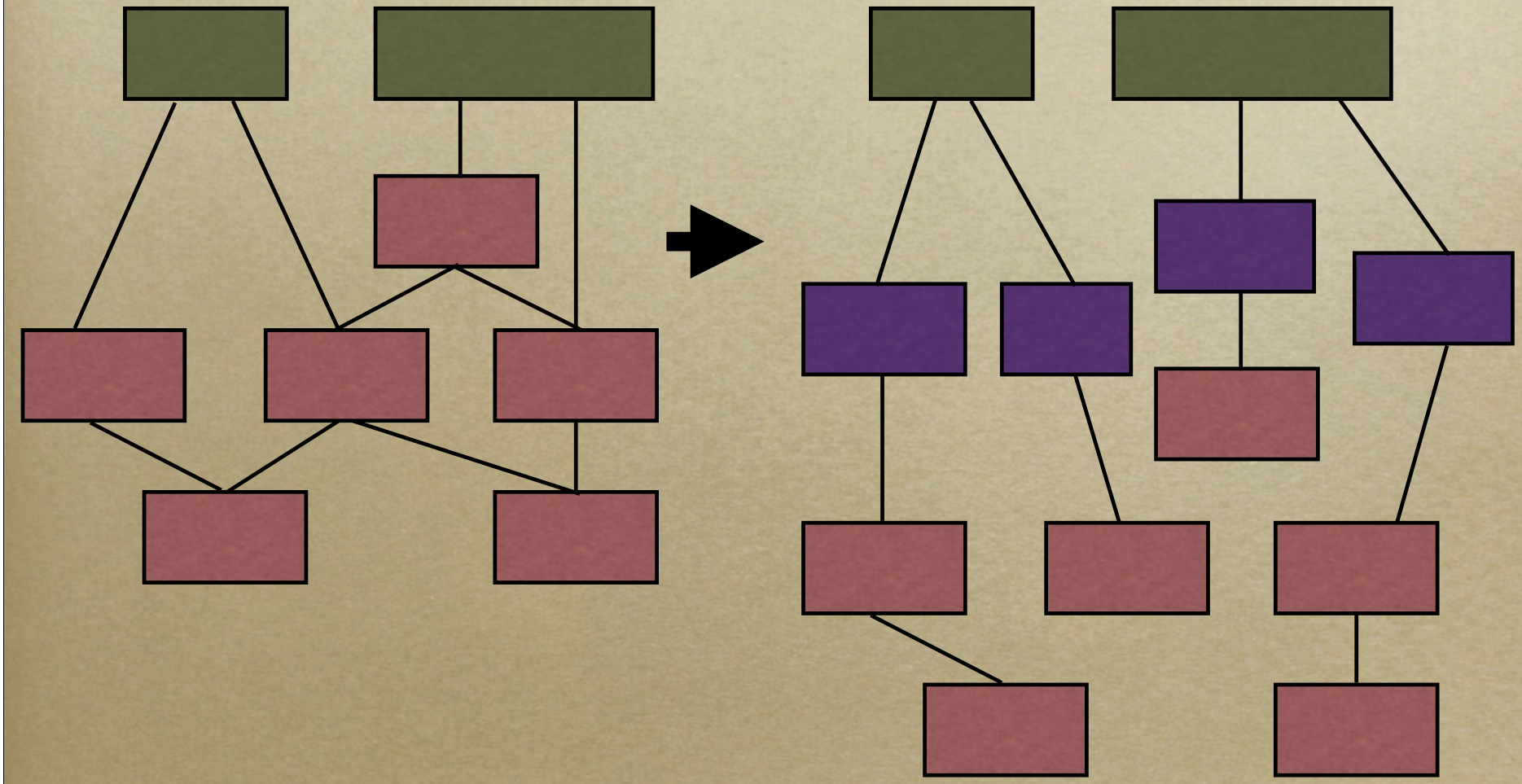
Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

Service Distributions

- Applications that push the envelope need arbitrary component composition
- Most don't push the envelope, though
- A service distribution is a set of services with compatible implementations and policies, presented as a unified API
  - An abstraction boundary

More Service Distributions



Example Distribution: OSKI

- A range of communication services
  - Broadcasting
  - Collection routing
  - Serial port
  - Addressed single-hop
- A power management policy
  - Turn service instances on and off
  - A service turns off iff all its instances are off

Effects

- Service distributions improve robustness
  - Preclude many unforeseen compositions
  - Can test services as a whole
- Service distributions make building applications easier
  - A real API, rather than a collection of independent components

Outline

- TinyOS
- Issues facing a second-generation OS
- T2 core structure
- Multi-layer abstractions
- Static binding and allocation
- Service distributions

The State of the Art Today

- Sensor networks are different
- Revisiting old problems and assumptions
  - Different resource constraints
  - Different design tradeoffs
  - Sometimes new problems, sometimes not
- Opening the doors to innovation
- Systems to support a huge range of requirements, applications, and abstractions

Components

- Innovation and exploration require flexible systems
  - Arbitrary boundaries
  - Arbitrary composition
- Current systems maximize flexibility
  - TinyOS: collections of components
- This flexibility sacrifices reliability
  - E.g, TinyOS -> SOS
  - Inherent tension between flexibility/reliability

The Hidden Cost

- Arbitrary flexibility prevents you from promising anything
  - Robustness, performance, etc.
- You can still build applications, but it's hard and time consuming
- T2: leverage five years of experience
  - Not all flexibility is needed
  - Can we trade off some for improved usability and performance?

The Research Questions

- Questions of grammar
  - How do we write a component?
  - How do we connect components?
  - How do we specify concurrency?
- Questions of style
  - How do we decompose a system into components (granularity, state sharing)?
  - How do we compose those components?
  - How do those components interact?

TinyOS 1.x
nesC

T2

Questions