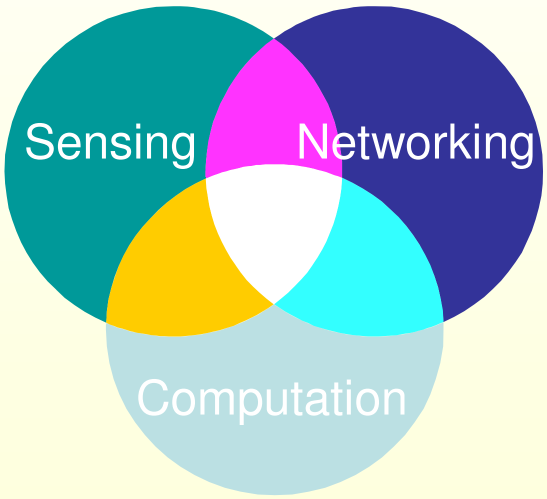
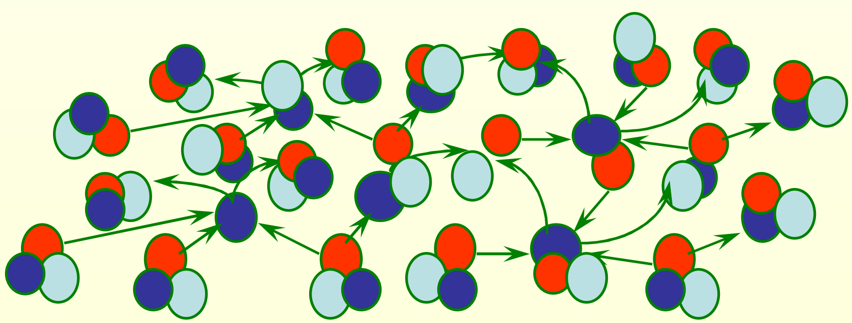
Sensors Network Software



Qing Fang
Stanford University



CS428

Outline

- Why is writing software for sensornet so hard?
- Programming platforms
  - TinyOS (Berkeley) – in details
  - Em* (UCLA) – very brief
- Discussion
  - Networking abstractions
  - Programming models

Embedded Networking Systems vs. the Internet –
Different set of Goals and Principles

The Internet:

- RFC 1958 ("Architectural Principles of the Internet") reads:

  "However, in very general terms, the community believes the goal is connectivity, the tool is the Internet Protocol, and the intelligence is end-to-end rather than hidden in the network… connectivity is its own reward, and is more valuable than any individual application"

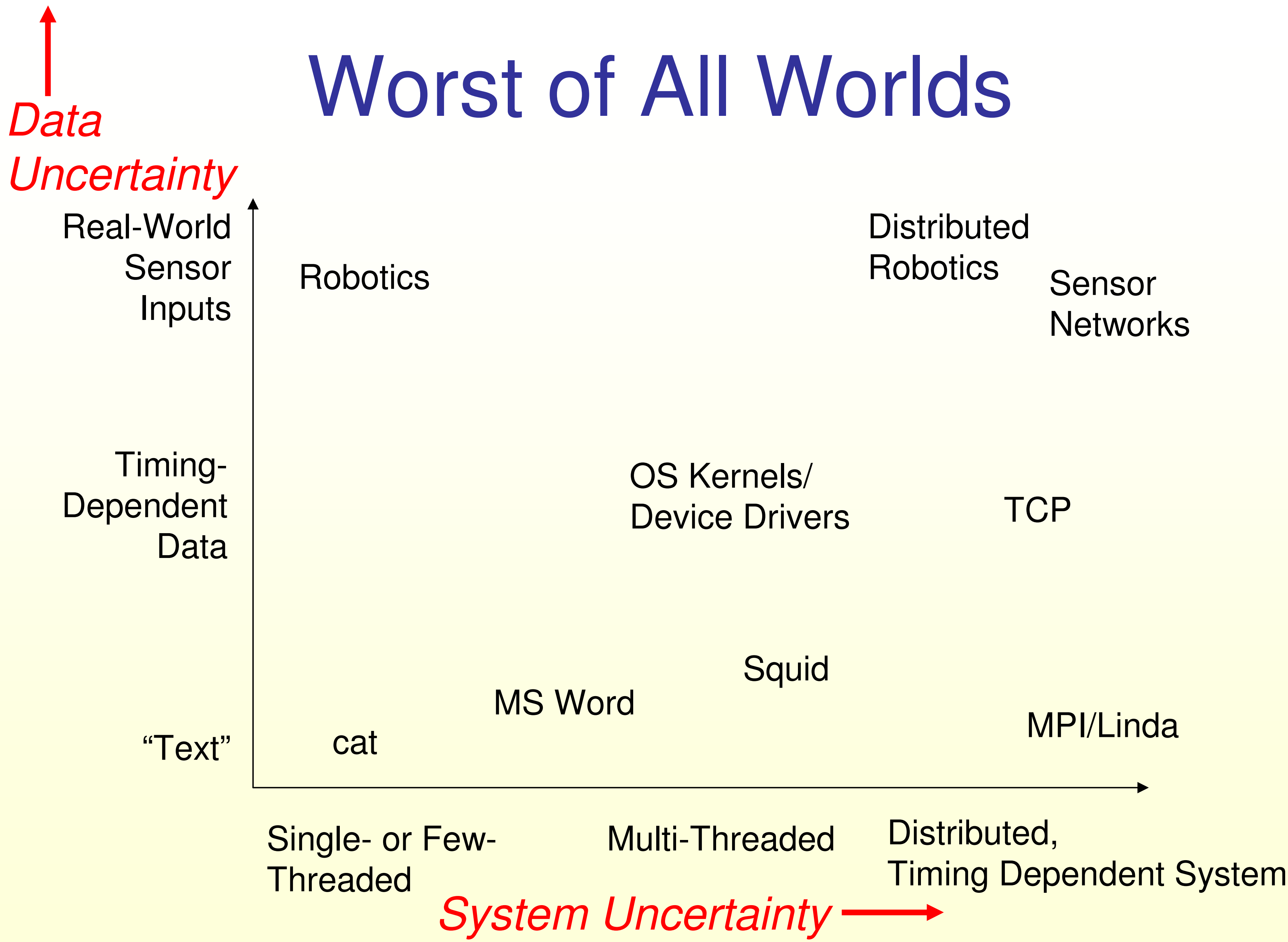
EmNets:

- The goal is application-specific collaboration among the nodes.
- In-network processing rather than end-to-end.

Software Challenges

- Uncertainty
  - System uncertainty
  - Data uncertainty
- Lack of a common architecture
- Energy constraints – may change over time

Worst of All Worlds

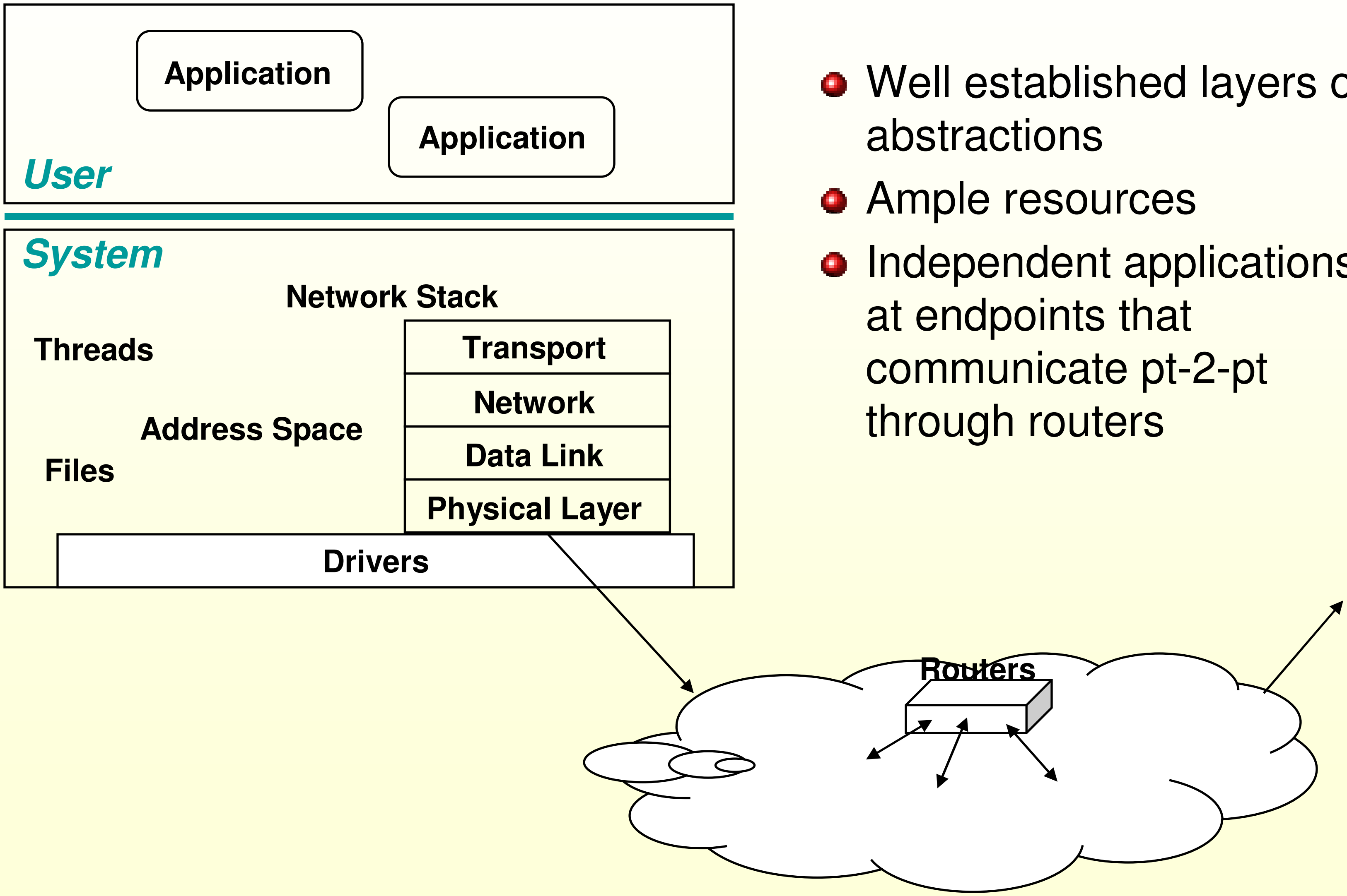


Source: Jeremy Elson, Microsoft Research

An Architecture

Is a set of principles that guide where functionality should be implemented along with a set of interfaces, functional components, protocols, and physical hardware that follows those guidelines.

Traditional System Architecture

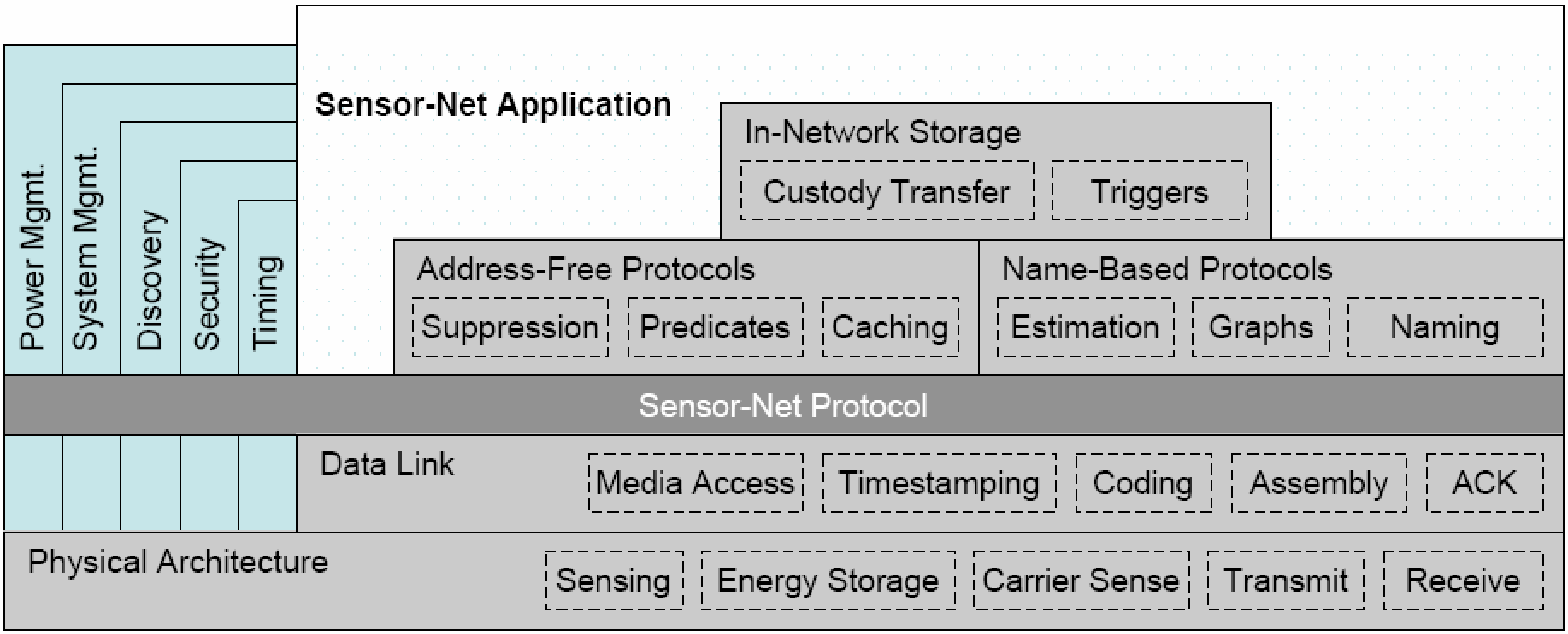


- Well established layers of abstractions
- Ample resources
- Independent applications at endpoints that communicate pt-2-pt through routers

A Sensor Network Architecture?

- No, don’t have one yet.
- We are still at the stage of extracting abstractions.
- But, we know what we want
  - Incorporates current generation of technology
  - Allows future innovation
  - Promotes interoperability

Sensornet Functional Layer Decomposition – Separation of Concerns



Source: David Culler, et al., Berkeley

TinyOS – Approach

- Does not define a particular system/user boundary nor a set of system services.

- Provides a framework for defining such boundaries and allows applications to select services and their implementations.

•128kB program flash

•512kB serial flash

TinyOS – Design Considerations

- Diversity in design and usage
- Robust
  - inaccessible, critical operation
- Concurrency intensive in bursts
  - streams of sensor data & network traffic
- Highly constrained resources
- Applications spread over many small nodes
  - self-organizing collectives
  - highly integrated with changing environment and network

efficient modularity

migration across HW/SW boundary

Need a framework for:

Resource-constrained concurrency

Need Application-specific processing that allows abstractions to emerge

TinyOS – Choices of Programming Primitives

- provide framework for concurrency and modularity
- never poll, never block
- interleaving flows, events, energy management

  allow appropriate abstractions to emerge

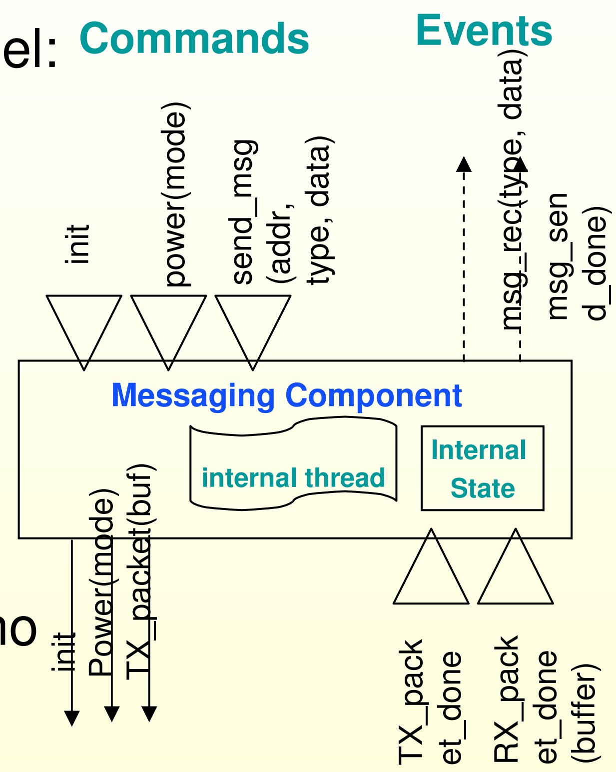
TinyOS Features

- Microthreaded OS (lightweight thread support)
- An event-driven concurrency model without blocking
- Two level scheduling structure
  - Long running *tasks* that can be interrupted by hardware *events*
- Modularity allows crossover of software components into hardware

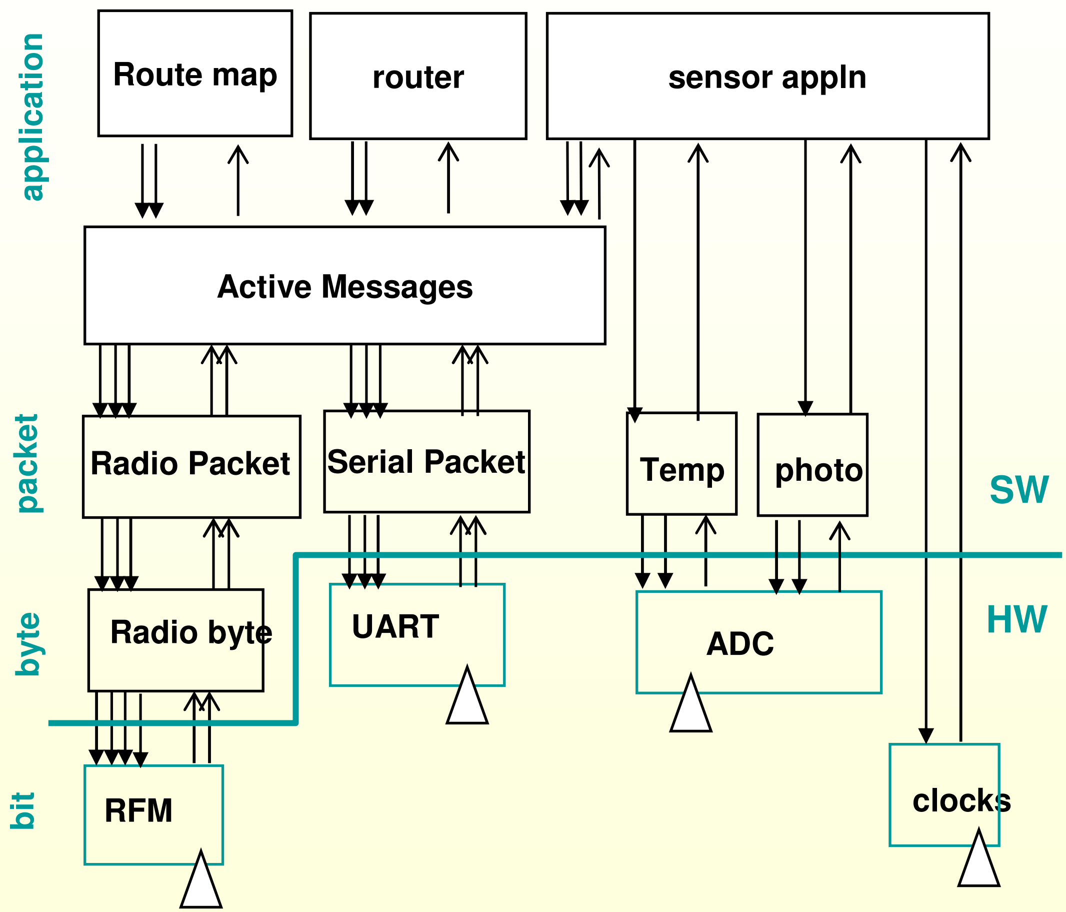
The following slides on TinyOS are from David Culler, et al., UC Berkeley

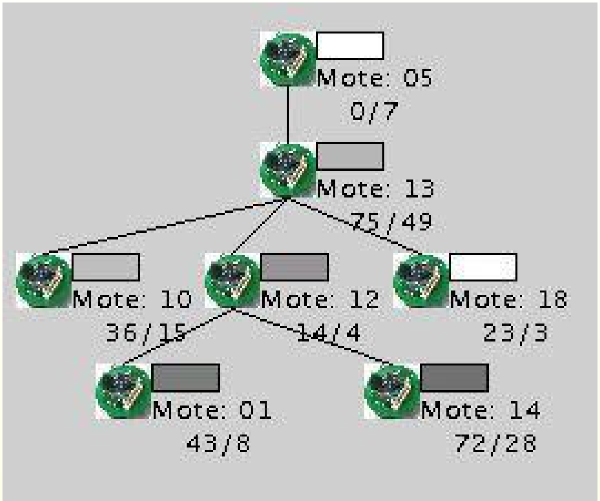
TinyOS Concepts

- Scheduler + Graph of Components
  - constrained two-level scheduling model: threads + events
- Component
  - Commands,
  - Event Handlers
  - Tasks (concurrency)
  - Frame (storage)
- Constrained Storage Model
  - frame per component, shared stack, no heap
- Very lean multithreading
- Layering



Application = Graph of Components





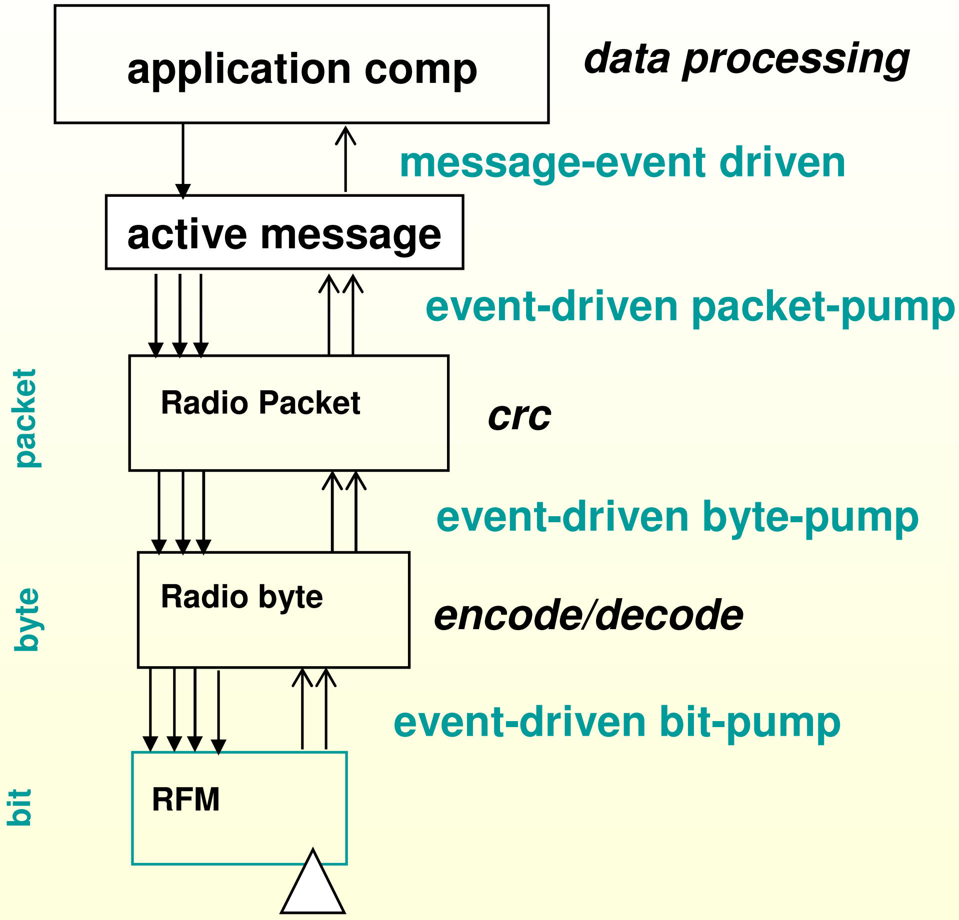
Example: ad hoc, multi-hop routing of photo sensor readings

3450 B code
226 B data

Graph of cooperating state machines on shared stack

TinyOS Execution Model

- commands request action
  - ack/nack at every boundary
  - call cmd or post task
- events notify occurrence
  - HW intrpt at lowest level
  - may signal events
  - call cmds
  - post tasks
- Tasks provide logical concurrency
  - preempted by events
- Migration of HW/SW boundary



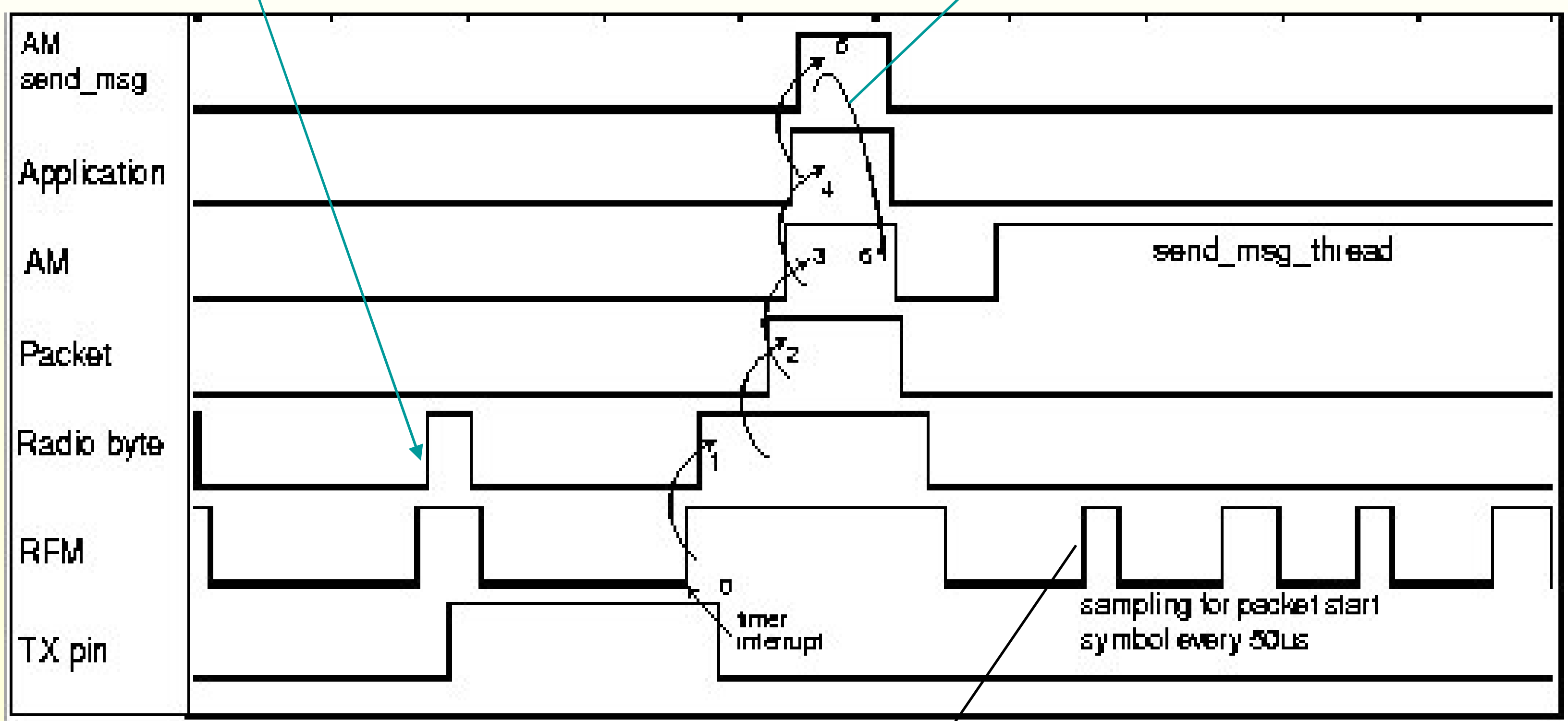
Dynamics of Events and Threads

bit event =>
end of byte =>
end of packet =>
end of msg send

thread posted to start

send next message

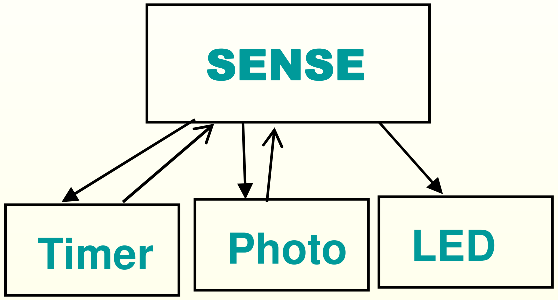
bit event filtered at byte layer



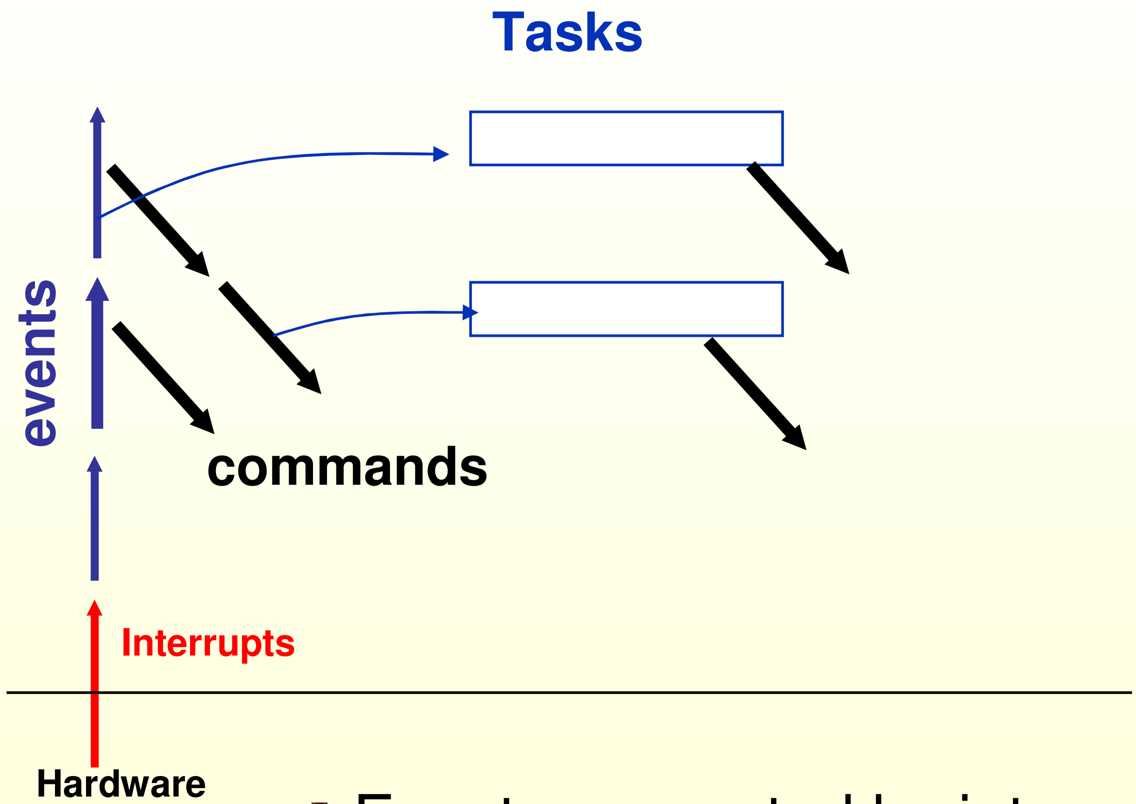
radio takes clock events to detect recv

Event-Driven Sensor Access Pattern

```
command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 200);
  }
event result_t Timer.fired() {
    return call sensor.getData();
  }
event result_t sensor.dataReady(uint16_t data) {
    display(data)
    return SUCCESS;
  }
```



TinyOS Execution Contexts

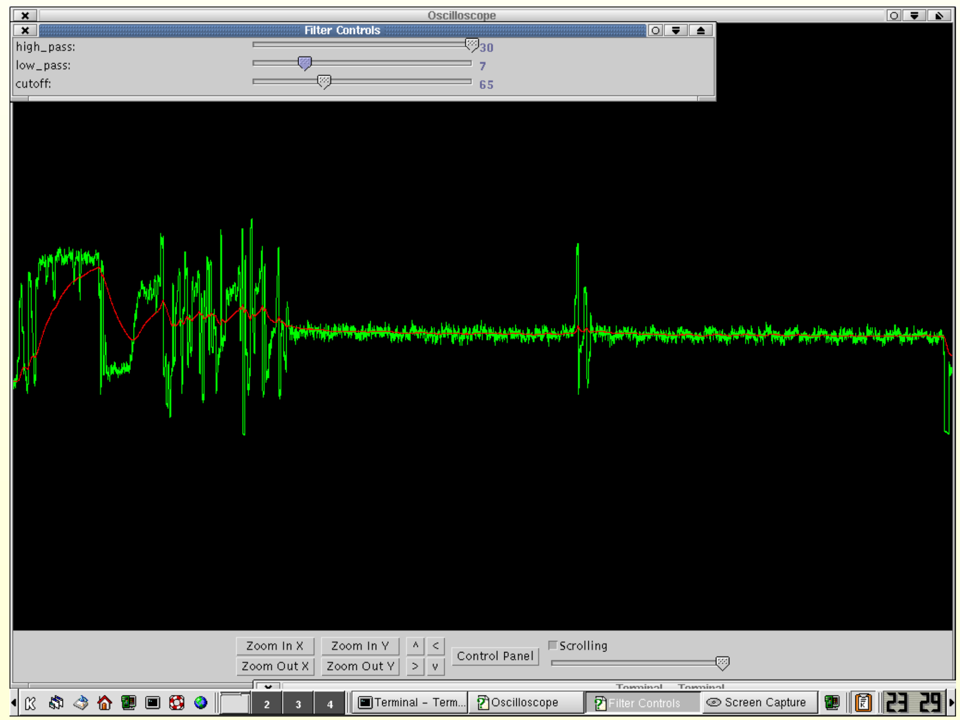


- Events generated by interrupts preempt tasks
- Tasks do not preempt tasks
- Both essential process state transitions

Typical application use of tasks

- event driven data acquisition
- schedule task to do computational portion

```
event result_t sensor.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
 }
task void processData() {
    int16_t i, sum=0;
    for (i=0; i ‹ maxdata; i++)
        sum += (rdata[i] ›› 7);
    display(sum ›› shiftdata);
 }
```



Tasks in low-level operation

- transmit packet
  - send command schedules task to calculate CRC
  - task initiated byte-level data pump
  - events keep the pump flowing
- receive packet
  - receive event schedules task to check CRC
  - task signals packet ready if OK
- byte-level tx/rx
  - task scheduled to encode/decode each complete byte
  - must take less time that byte data transfer

Task Scheduling

- Currently simple fifo scheduler
- Bounded number of pending tasks
- When idle, shuts down node except clock
- Uses non-blocking task queue data structure
- Simple event-driven structure + control over complete application/system graph

Tiny Active Messages

- Sending
  - Declare buffer storage in a frame
  - Request Transmission
  - Name a handler
  - Handle Completion signal
- Receiving
  - Declare a handler
  - Firing a handler
- Buffer management
  - strict ownership exchange
  - tx: done event => reuse
  - rx: must rtn a buffer

Sending a message

```
bool pending;
struct TOS_Msg data;
command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)data.data;
    if (!pending) {
         pending = TRUE;
         message->val = value;
         message->src = TOS_LOCAL_ADDRESS;
         if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))
              return SUCCESS;
          pending = FALSE;
    }
    return FAIL;
}
```

destination (TOS_BCAST_ADDR)

length (sizeof(IntMsg))

- Refuses to accept command if buffer is still full or network refuses to accept send command
- User component provide structured msg storage

Send done event

```
 event result_t IntOutput.sendDone(TOS_MsgPtr msg,
                                                result_t success)
  {
    if (pending && msg == &data) {
        pending = FALSE;
        signal IntOutput.outputComplete(success);
      }
    return SUCCESS;
  }
}
```

Receive Event

```
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
        IntMsg *message = (IntMsg *)m->data;
        call IntOutput.output(message->val);
        return m;
}
```

- Active message automatically dispatched to associated handler
  - knows the format, no run-time parsing
  - performs action on message event
- Must return free buffer to the system
  - typically the incoming buffer if processing complete

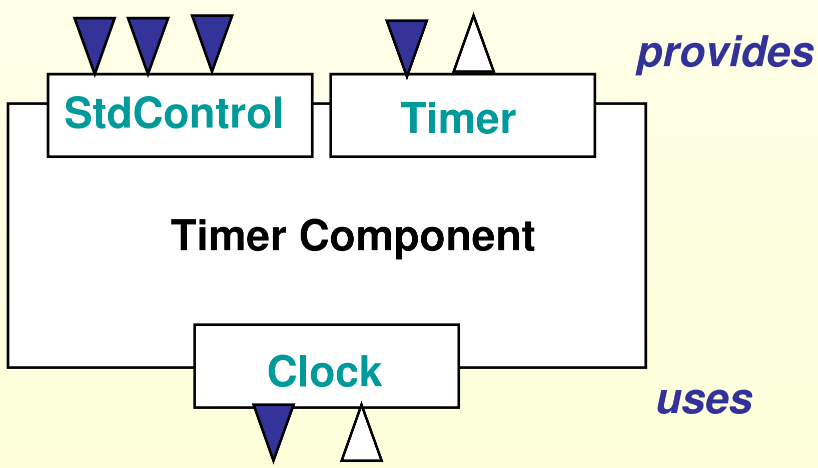
Programming TinyOS

- TinyOS 1.0 is written in an extension of C, called nesC
- Applications are too!
  - just additional components composed with the OS components
- Provides syntax for TinyOS concurrency and storage model
  - commands, events, tasks
  - local frame variable
- Rich Compositional Support

Composition

- A component specifies a set of *interfaces* by which it is connected to other components
  - provides a set of interfaces to others
  - uses a set of interfaces provided by others
- Interfaces are bi-directional
  - include commands and events
- Interface methods are the external namespace of the component

```
provides
    interface StdControl;
    interface Timer:
uses
    interface Clock
```



Components

- Modules
  - provide code that implements one or more interfaces and internal behavior
- Configurations
  - link together components to yield new component
- Interface
  - logically related set of commands and events

StdControl.nc

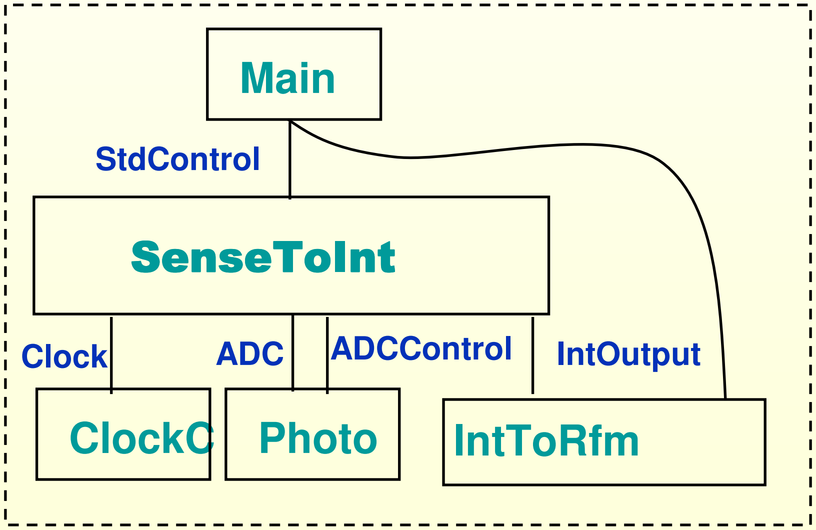
```
interface StdControl {
      command result_t init();
      command result_t start();
      command result_t stop();
}
```

Clock.nc

```
interface Clock {
   command result_t setRate(char interval, char scale);
   event result_t fire();
}
```

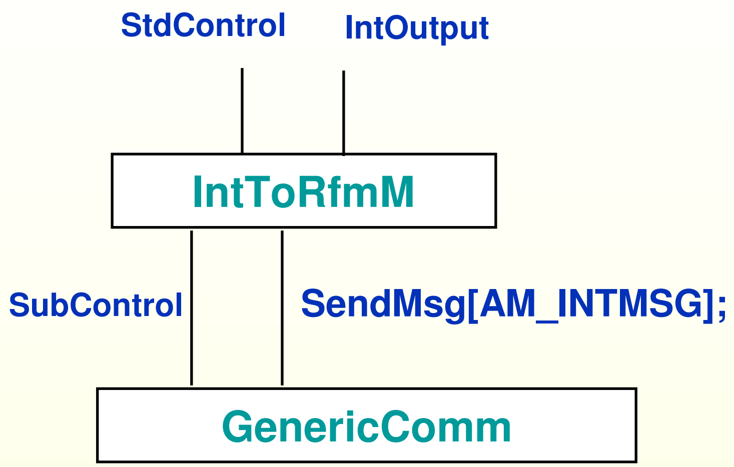
Example top level configuration

```
configuration SenseToRfm {
// this module does not provide any interface
}
implementation
{
  components Main, SenseToInt, IntToRfm, ClockC, Photo as
Sensor;

  Main.StdControl -> SenseToInt;
  Main.StdControl -> IntToRfm;

  SenseToInt.Clock -> ClockC;
  SenseToInt.ADC -> Sensor;
  SenseToInt.ADCControl -> Sensor;
  SenseToInt.IntOutput -> IntToRfm;
}
```



Nested configuration

```
includes IntMsg;
configuration IntToRfm
{
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  components IntToRfmM, GenericComm as Comm;

  IntOutput = IntToRfmM;
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.SubControl -> Comm;
}
```

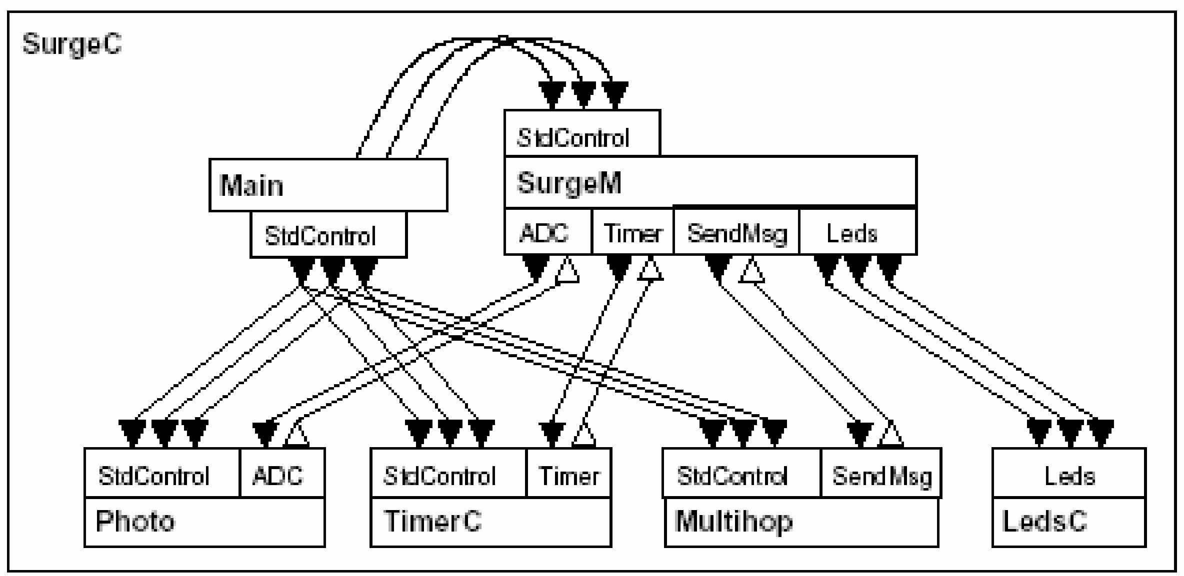


IntToRfm Module

```
includes IntMsg;

module IntToRfmM
{
  uses {
    interface StdControl as SubControl;
    interface SendMsg as Send;
  }
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  bool pending;
  struct TOS_Msg data;

  command result_t StdControl.init() {
    pending = FALSE;
    return call SubControl.init();
  }
```

```
command result_t StdControl.start()
  { return call SubControl.start(); }
command result_t StdControl.stop()
   { return call SubControl.stop(); }

command result_t IntOutput.output(uint16_t
value)
  {
    ...
    if (call Send.send(TOS_BCAST_ADDR,
         sizeof(IntMsg), &data)
         return SUCCESS;
     ...
  }

event result_t Send.sendDone(TOS_MsgPtr
msg, result_t success)
{
```

A Multihop Routing Example



Supporting HW evolution

- Distribution broken into
  - apps: top-level applications
  - lib: shared application components
  - system: hardware independent system components
  - platform: hardware dependent system components
- Component design so HW and SW look the same
- HW/SW boundary can move up and down with minimal changes

TinyOS tools

- TOSSIM: a simulator for tinyos programs
- ListenRaw, SerialForwarder: java tools to receive raw packets on PC from base node
- Oscilloscope: java tool to visualize (sensor) data in real time
- Memory usage: breaks down memory usage per component (in *contrib*)
- Peacekeeper: detect RAM corruption due to stack overflows (in *lib*)
- Stopwatch: tool to measure execution time of code block by timestamping at entry and exit
- Makedoc and graphviz: generate and visualize component hierarchy
- Surge, Deluge, SNMS, TinyDB

TinyOS Limitations

- Static allocation allows for compile-time analysis, but can make programming harder
- No support for heterogeneity
  - Support for other platforms (e.g. stargate)
  - Support for high data rate apps (e.g. acoustic beamforming)
  - Interoperability with other software frameworks and languages
- Limited visibility
  - Debugging
  - Intra-node fault tolerance
- Robustness solved in the details of implementation
  - nesC offers only some types of checking

Em*

- Software environment for sensor networks built from Linux-class devices
- Claimed features:
  - Simulation and emulation tools
  - Modular, but not strictly layered architecture
  - Robust, autonomous, remote operation
  - Fault tolerance within node and between nodes
  - Reactivity to dynamics in environment and task
  - High visibility into system: interactive access to all services

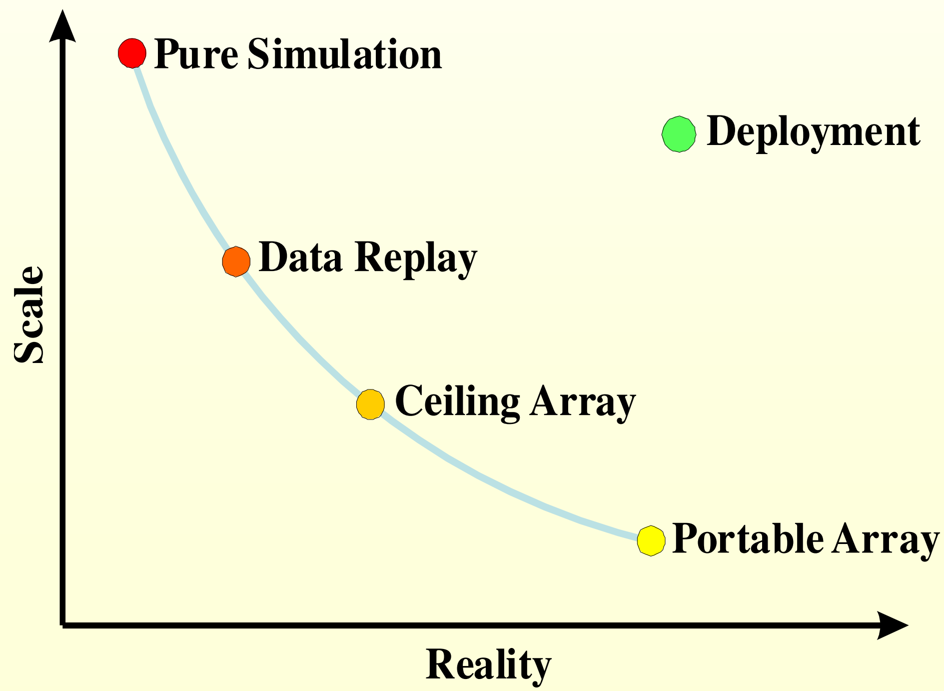
Contrasting Emstar and TinyOS

- Similar design choices
  - programming framework
    - Component-based design
    - “Wiring together” modules into an application
  - event-driven
    - reactive to “sudden” sensor events or triggers
- Differences
  - hardware platform-dependent constraints
    - Emstar: Develop without optimization
    - TinyOS: Develop under severe resource-constraints
  - operating system and language choices
    - Emstar: easy to use C language, tightly coupled to linux
    - TinyOS: an extended C-compiler (nesC), an OS by itself

Em* Transparently Trades-off Scale vs. Reality

Em* code runs transparently at many degrees of “reality”:

high visibility debugging before low-visibility deployment



Other Platforms

- SOS – UCLA
- Contiki – Swedish Institute of Computer Science
- Virtual machines (Maté) −UC Berkeley

Go Back to the Architecture Challenge

We need higher level abstractions!

Why?

- They let you reason about software at a higher level.
- They let software interoperate better.
  - Compact
  - Consistent
  - Reuse

And this calls for…

Towards Higher Level Abstractions

- Better understanding of the applications
- More efficient and effective algorithms
  - Designing local rules to cause global behavior is hard

The Emerging Networking Abstractions

- Singer hop communication – active message
- Multi-hop communication
  - Tree based routing
  - Directed diffusion
  - Broadcast
  - Epidemic protocols
  - Landmark based routing? ☺
- Power management
- Time synchronization

Mechanism vs. policy

Programming Models

- TinyOS is no fun to program
  - Split-phase operation   A logically blocking sequence must be written in a state-machine style.

- State Machine Model? (ETH)*
- Token Machine Model? (MIT)*

* In proceedings of IPSN, 2005

The End