Building Sensor Networks with TinyOS

David Culler, Phil Levis, Rob Szewczyk, Joe Polastre
University of California, Berkeley
Intel Research Berkeley

http://www.cs.berkeley.edu

Technology Push

- CMOS miniaturization
  - 1 M trans/s => tiny (~mm), inexpensive processing and storage
  - 1-10 mW active, 1 pW passive
- Micro-sensors (MEMS, Materials, Circuits)
  - acceleration, vibration, gyroscope, tilt, magnetic, heat, motion, pressure, temp, light, moisture, humidity, barometric
  - chemical (CO, CO2, radon), biological, micro-radar, ...
  - actuators too (mirrors, motors, smart surfaces, micro-robots)
- Communication
  - short range, low bit-rate, CMOS radios (1-10 mW)
- Power
  - batteries remain primary storage (1,000 mW/mm²), fuel cells 10x
  - solar (10 mW/cm² 6.1 mW indoors), vibration (~1W/gm), flow
- 1 cm³ battery => 1 year at 10 msgs/sec

Application Pull

- UAV drops 10 nodes along road,
  - hot-water pipe insulation for package
- Nodes self-configure into linear network
- Synchronize (to 1/32 s)
- Calibrate magnetometers
- Each detects passing vehicle
- Share filtered sensor data with 5 neighbors
- Each calculates estimated direction & velocity
- Share results
- As plane passes by,
  - joins network
  - upload as much of missing dataset as possible from each node when in range
- ~7.5 KB of code!
- While servicing the radio in SW every 50 us!

Example: Environment Monitoring

- Canonical 'patch' net architecture
- live & historical readings
  www.gratebucketland.net
- 43 nodes, 7/13-11/18
- above and below ground
- light, temperature, relative humidity, albedo & occupancy data, at 1 minute
- >1 million measurements
  - Best nodes ~90,000
- 3 major maintenance events
- power management and packaging in harsh environments
  - 20 – 100 degrees, rain, wind
- power mgmt and interplay with sensors and environment

The Challenges

Monitoring & Managing Spaces and Things

Miniature, low-power connections to the physical world
An Operating System for Tiny Devices embedded in the Physical World

Characteristics of Network Sensors
- Small physical size and low power consumption
- Concurrency-intensive operation
- Limited Physical Parallelism and Controller Hierarchy
  - primitive direct-to-device interface
  - Asynchronous and synchronous devices
- Diversity in Design and Usage
  - application specific, not general purpose
  - huge device variation
- Robust Operation
  - numerous, unattended, critical
- Efficient Layering

Traditional Systems
- Well established layers of abstractions
- Strict boundaries
- Independent Applications at endpoints communicate pt-pt through routers
- Well attended

by comparison ...
- Highly Constrained resources
  - processing, storage, bandwidth, power
- Applications spread over many small nodes
  - self-organizing Collectives
  - highly integrated with changing environment and network
  - communication is fundamental
- Concurrency intensive in bursts
  - streams of sensor data and network traffic
- Robust
  - inaccessible, critical operation
- Unclear where the boundaries belong

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

Application = Graph of Components

Example: ad hoc, multi-hop routing of photo sensor readings
3400 B code
226 B data
TOS 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

TinyOS Commands and Events

- command CmdName(args) {
  status = call CmdName(args);
  return status;
}

- event EvtName(args) {
  status = signal EvtName(args);
  return status;
}

TinyOS Execution Contexts

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

TASKS

- provide concurrency internal to a component
  - longer running operations
- are preempted by events
- able to perform operations beyond event context
- may call commands
- may signal events
- not preempted by tasks

Typical application use of tasks

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

void processData() {
    uint16_t i, sum=0;
    for (i=0; i < maxData; i++)
        sum += (rdData[i] >> 7);
    display(sum, shiftData);
}
```

Example: Radio Byte Operation

- Pipelines transmission – transmits single byte while encoding next byte
- Trades 1 byte of buffering for easy deadline
- Separates high level latencies from low level real-time requirements
- Encoding Task must complete before byte transmission completes
- Decode must complete before next byte arrives
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

- 2c component
  - 2c bus has long suspend operations
  - tasks used to create split-phase interface
  - events can proceed during bus transactions

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
  - instead of complex task priorities and IPC

Tiny Active Messages

- Sending
  - Declare buffer storage in a frame
  - Request Transmission
  - Name a handler
  - Handle Completion signal

- Receiving
  - Declare a handler
  - Fixing a handler
  - Automatic
  - Behaves like any other event

- Buffer management
  - Strict ownership exchange
  - Tx: done event -> reuse
  - Rx: must rm a buffer

Sending a message

```c
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;
    IntOutput.output(TOS_LOCAL_ADDRESS);
    if (call Send.send(TOS_BCAST_ADDR) [sizeof(IntMsg] sdata) {
      return SUCCESS;
    } pending = FALSE;
    return FAIL;
  }
}
```

- Refuses to accept command if buffer is still full or network refuses to accept send command

- User component provide structured msg storage

Receive Event

```c
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

Send done event

```c
event result_t IntOutput.sendDone(TOS_MsgPtr seg, result_t success) {
  if (pending && seg == &data) {
    pending = FALSE;
    IntOutput.outputComplete(success);
    return SUCCESS;
  }
  Send done event fans out to all potential senders
  Originator determined by match
  - free buffer on success, retry or fail on failure
  Others use the event to schedule pending communication
```
Maintaining Scheduling Agility

- Need logical concurrency at many levels of the graph
- While meeting hard timing constraints
  - sample the radio in every bit window

⇒ Retain event-driven structure throughout application
⇒ Tasks extend processing outside event window
⇒ All operations are non-blocking

The Complete Application

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

Component

- 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

Example top level configuration

```
configuration SenseToRFCm {
    // this module does not provide any interface
}
implementation {
    components Main, SenseToPoint, IntToRFCm, ClockC, Photo as Sensor;
    Main.StdControl -> SensePoint;
    Main.StdControl -> IntToRFCm;
    SenseToPoint.Clock -> ClockC;
    SenseToPoint.SequenceControl -> Sensor;
    SenseToPoint.IntInputControl -> IntToRFCm;
}
```

Nested configuration

```
configuration IntToRFCm {
    provides {
        interface StdOutput, interface StdControl;
    }
}
implementation {
    components IntToRFCm, GenericComm as COnneC;
    IntInput = IntToRFCm;
    IntOutput = IntToRFCm;
    StdControl = IntRFCm;
    StdOutput = IntRFCm;
    StdOutput.Send = COnneC.SendMsg[AM_INTMSG];
    StdControl.SubControl = COnneC;
}
```
IntToRfm Module

```c
#include "msg.h"
module IntToRfm {
  struct IntMsg data;
  module IntToRfmModule {
    void InterfaceSendMsg (u8 addr) {
      if (msg.Send(A2S_ADDR, addr, ControlMsg, vbox)) return SUCCESS;
      send_result_t msgSend(Done(TOS_MsgPtr msg, vbox));
    }
    int InterfaceOutput {
      return call_SubControl_init();
      pendin = FALSE;
    }
  }
  interface InterfaceStdControl;
  interface SendMsg as Send;
  interface StdControl as SubControl;
  int TargetPlatform: TOSSIM
  • Distribution broken into
    – apps: top-level applications
    – lib: shared application components
    – system: hardware independent system components
    – platform: hardware dependent system components
      • includes HPLs and hardware.h
  • Component design so HW and SW look the same
    – example: temp component
      • may abstract particular channel of ADC on the microcontroller
      • may be a SW UART protocol to a sensor board with digital
        sensor or ADC
  • HW/SW boundary can move up and down with minimal changes

Sample Components

- Communication
  - Radio, UART, I2C of various flavors
- Timing
  - Timer, Clock
- Sensors
  - voltage, photo, light
- Busses
  - I2C, SPI
- Storage
  - nand, logger
- Energy management
  - snooze

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
  - separation of definition and linkage
  - robustness through narrow interfaces and reuse
  - interpositioning
- Whole system analysis and optimization

Scalable Simulation Environment

- target platform: TOSSIM
  - whole application compiled for host native instruction set
  - event-driven execution mapped into event-driven simulator
    machinery
  - storage model mapped to thousands of virtual nodes
- radio model and environmental model plugged in
  - bit-level fidelity
- Sockets = basestation
- Complete application
  - including GUI

Where to go for more?

- http://www.tinyos.net/tos/
- http://sourceforge.net/projects/tinyos/

Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David
Culler, Kristopher Pister. System architecture directions for
network sensors. ASPLOS 2000.
- David E. Culler, Jason Hill, Philip Buonadonna, Robert
Szewczyk, and Alec Woo. A Network-Centric Approach to