TinyOS

Jan S. Rellermeyer
jrellermeyer@student.ethz.ch

Overview

- Motivation
- Hardware
- TinyOS Architecture
- Component Based Programming
- nesC
- TinyOS Scheduling
- Tiny Active Messaging
- TinyOS Multi Hop Routing
- TinyDB
- Conclusion

Motivation: Sensor Networks

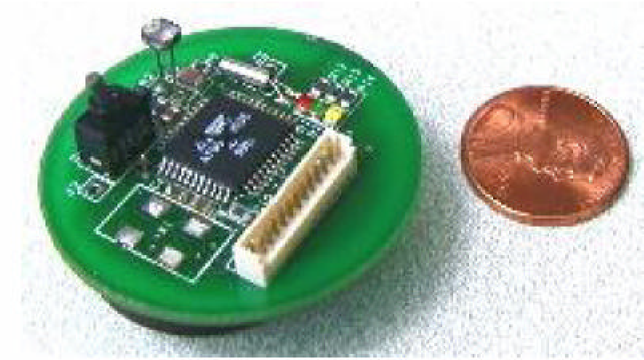
- Low cost, low power, small in size, communicate in short distances
- Large number of sensors, densely deployed inside of close to the phenomenon
- Position not predetermined, can be randomly distributed
- Self-organising distributed ad-hoc networks, nodes might fail, move etc.
- Cooperative effort, instead of sending raw data, do some local computation and transmit only required and partially processed data

Sensor Networks (contd.)

- Network characteristics
  - Total number of nodes is unknown, might be very large
  - Topology is unknown, can change in time
  - Densely deployed: point to point communication ?
  - Nodes are limited in power, computational capacities, memory, radio range
  - Maybe even no global ID‘s due to large amount of overhead and large number of sensors (Smart Dust ?)
- Sensing, processing and transmitting data
- Applications: Collecting data in environment
- Or can be used to simulate general MANETs

Hardware – Mica Mote

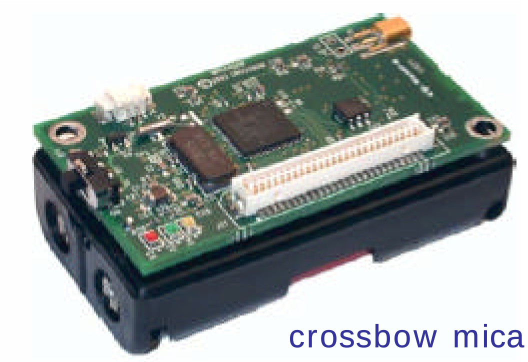
- Developed at University of Berkeley, CA
- CPU: Atmel AVR ATmega128L µcontroller
  - 128 kB flash ROM, 4 kB RAM, 4 kB EEPROM
  - Running at 4 MHz, 3.0 V
  - Power management: *idle* (CPU asleep), *power down* (only watchdog and interrupt for wakeup), *power save* (power down plus timer)
  - Harvard style 16 bit address space
  - 8 bit RISC machine
  - 32 8-bit registers
  - Highly orthogonal instruction set



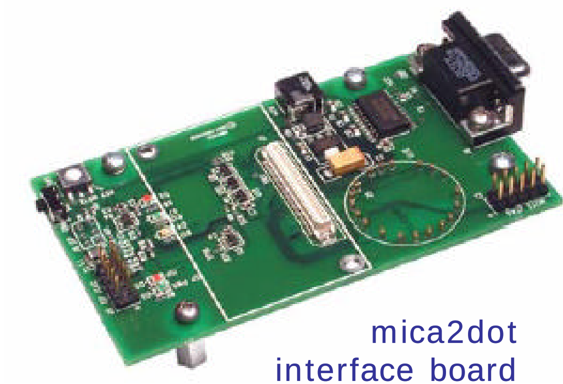
Berkeley mica2dot

Hardware – Mica Mote (contd.)

- Radio: Chipcon CC 1000
  - UHF transceiver (300 MHz – 1 GHz)
  - FSK modulation, up to 76.8 kBaud
  - No buffering (!)
  - Range: 100s of feet
- IO:
  - Photo sensor + internal A/D converter
  - 3 output LEDs
  - Different sensor boards via I²C Bus (up to 8 devices)
- UART serial port controller (via interface board)



crossbow mica



mica2dot interface board

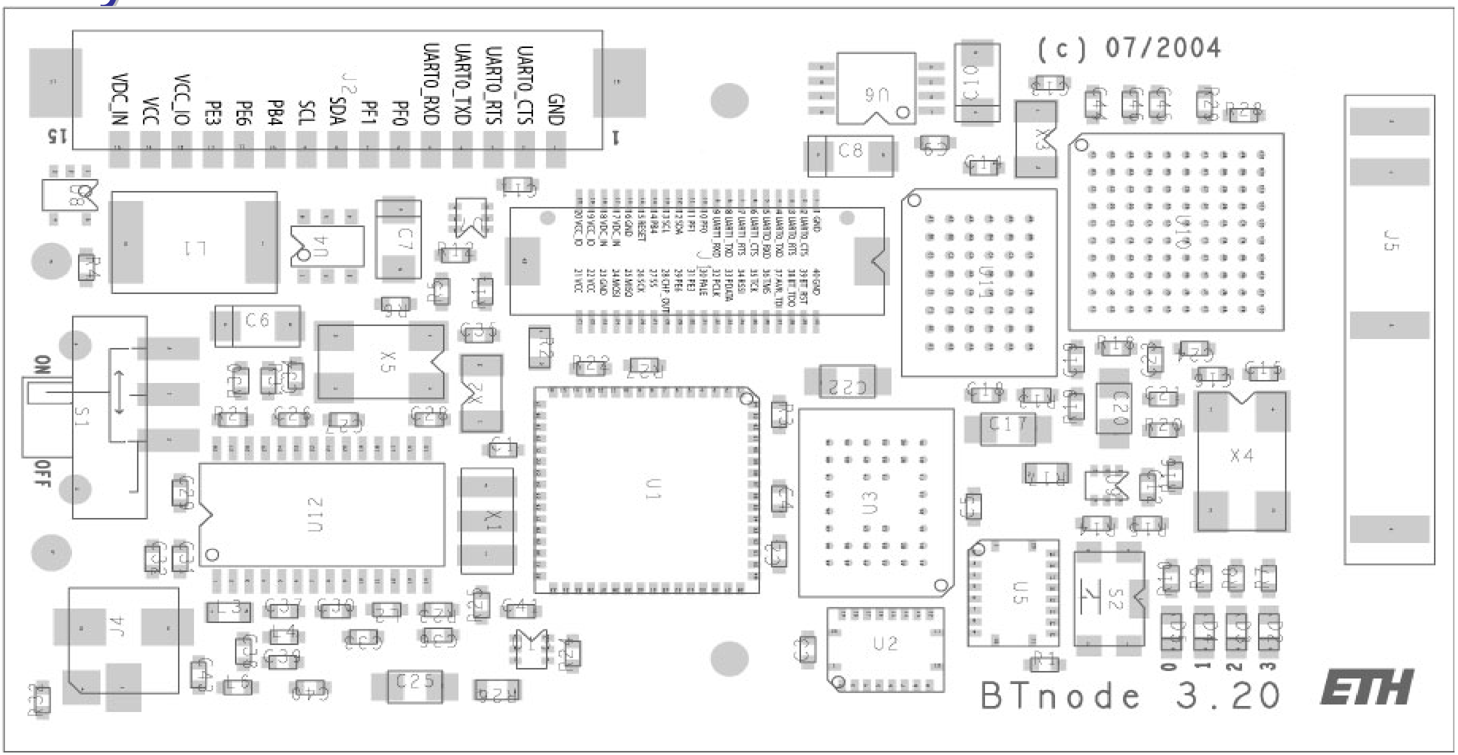
Hardware - BTnode

ETHZ BTnode

- Developed at TIK (D-ITET) and DSG (D-INFK) ETH Zürich
- CPU: Also Atmel AVR ATmega128L
  - 4 kB EEPROM, 64 kB SRAM, 128 kB Flash
- Radio: Zeevo ZV4002 Bluetooth radio
  - Supports up to 4 independent piconets and 7 slaves
- Low power radio: Mica2 Mote compatible Chipcon CC1000
- Can run with TinyOS (portation by University of København) or BTnut system software (ETHZ)

Hardware – BTnode (contd.)

Build your own BTnode ...



Graphic by TIK, E

Or get one from Art of Technology, Zurich (~100 $)

System constraints

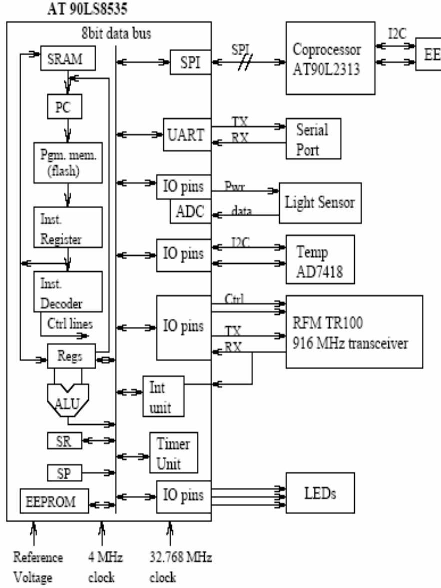
- Power consumption:

Component	Active (mA)	Idle (mA)	Inactive (µA)
MCU core (AT90S8535)	5	2	1
MCU pins	1.5	-	-
LED	4.6 each	-	-
Photocell	0.3	-	-
Radio (RFM TR 1000)	12 tx, 4.5 rcv	-	-
Temperature (AD7416)	1	0.6	1.5
CoPro (AT90LS2343)	2.4	0.5	1
EEPROM (24LC256)	3	-	1

- Philosophy for OS: “sleep, wake up, do work, sleep”

System constraints (contd.)

- Ordinary computing devices use a stack-based threaded model
  - Each process / thread has it’s own text, data and stack
  - OS has scheduler to switch the context periodically
- Not enough resources to do this on motes:
  - QNX context switch: 2400 cycles on x86
  - pOSEK context switch: > 40 µs
- What we want is:
  - Single stack
  - Single execution context
  - Handle physical parallelism (rfm, sensors)



Graphic: J. Hill

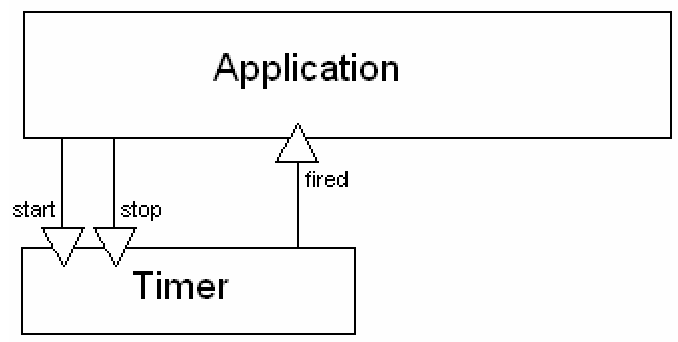
TinyOS architecture

- Small footprint: fits in 178 Bytes of memory
- Event based instead of threaded architecture
  - Propagates events in time it takes to copy 1.25 Bytes
  - Switches context in the time to copy 6 Bytes of Mem.
  - Used to call a higher level from a lower
- Component Based
- Radio and Clock have interrupts
- Concurrency with Tasks
  - Tasks are intended to do arbitrary computation, Events and Commands do state transitions
  - Tasks are queued, on empty queue, CPU sleeps

Component Based Programming

- Paradigm: Separate API from Implementation and encapsulate large quantum of functionality
  - Example: OSGi – Dynamic Bundle Architecture
- General Approach: Use Interfaces at higher level, lower level is the implementation
- TinyOS: bidirectional interfaces
  - Commands are specified at higher level and implemented at lower level
  - Events are specified at lower level and implemented at higher level
  - Events comparable to callback functions

Commands and Events



- Example:
  start and *stop* are commands
  fired is a event invoked by Timer
  The interface specifies:
  Timer must implement *start* and *stop*, Application must implement *fired*
- Commands:
  - Non blocking requests made to lower-level component. Deposits parameters to its frame and conditionally schedule a task for execution
- Events:
  - Event Handlers handle events from lower-level components. Event handlers can signal higher-level events, posts tasks or call lower-level commands

nesC

- TinyOS is based on nesC, a dialect of C
  - Imperative, C-like on low level
  - More declarative at top level
  - Very modular
- Programs are build from components, that are either modules or configurations
- Modules implement interfaces with functions (command and events)
- Configurations connect interfaces together (wiring)

HelloWorld for motes: HelloM.nc

```
module HelloM {
   provides {
         interface StdControl;
   }
   uses {
         interface Timer;
         interface Leds;
   }
}
```

- StdControl is the interface for all executables:
  - Commands result_t init(), result_t start() and result_t stop()
  - Semantic: init * (start | stop)*
  - init is normally used to power up hardware

HelloM.nc (2)

```
implementation {

  command result_t StdControl.init() {
      call Leds.init();
      return SUCCESS;
  }

  command result_t StdControl.start() {
      return call Timer.start(TIMER_ONE_SHOT, 1000);
  }

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

HelloM.nc (3)

```
  event result_t Timer.fired() {
      call Leds.redOn();
      call Leds.greenOn();
      call Leds.yellowOn();
      return SUCCESS;
  }

} // implementation
```

- Interfaces implemented, now the wiring ...

Hello.nc

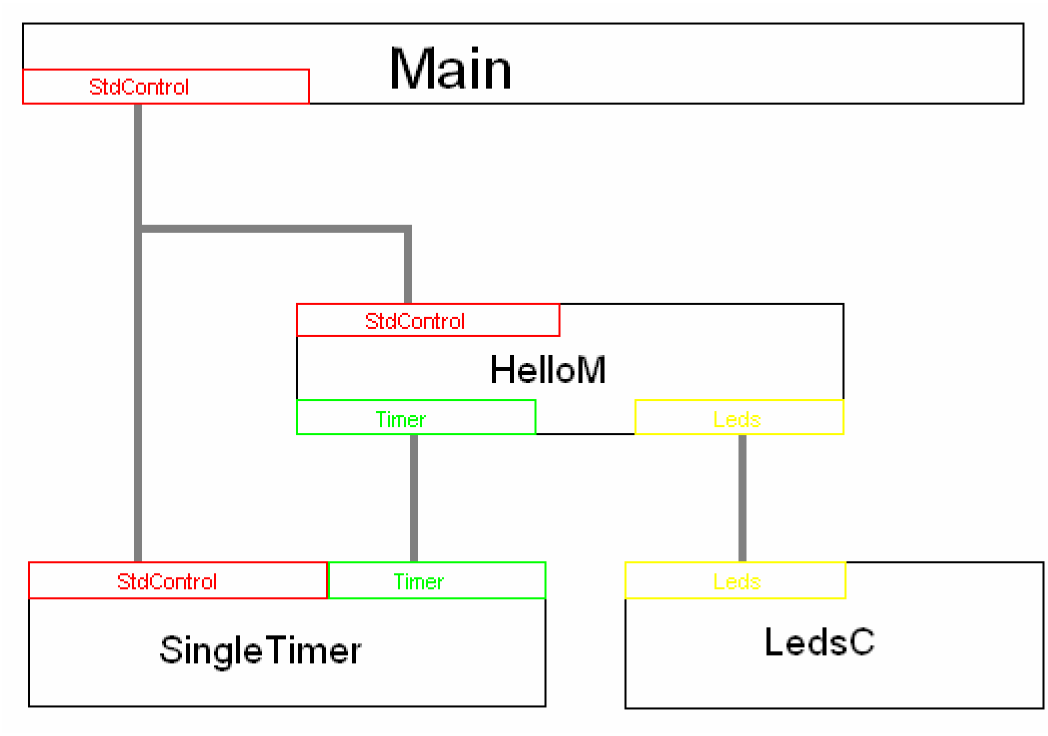
```
configuration Hello {
}
implementation {
   components Main, HelloM, SingleTimer, LedsC;

   Main.StdControl -> HelloM.StdControl;
   Main.StdControl –> SingleTimer;
   HelloM.Timer -> SingleTimer.Timer;
   HelloM.Leds -> LedsC;
}
```

- Interfaces are connected, HelloM provides StdControl and uses Timer and Leds.

  connect HelloM to Main and SingleTimer and LedsC

Hello.nc (2)

- The picture:



Component Based Programming revisited

- Is SingleTimer software or hardware and LedsC ?
  - Does not matter, components can be moved from software to hardware to increase speed without changing the applications.
  - Think of network routing algorithms in hardware, significant speedup
- Only interfaces are known, implementations can change
- Kind of “design by contract”

TinyOS

- TinyOS is a runtime environment for nesC running on mote hardware
  - Performs some resource management
  - Selected components are linked into the program at compile time
- TinyOS provides components for:
  - AD conversion
  - Cryptography
  - File System
  - LED control
  - Memory allocation
  - Data logging
  - Random numbers
  - Routing
  - Serial Communication
  - Timers
  - Watchdog
  - Sensor Board Input

TinyOS Scheduling

- FIFO Scheduler with queue length 7
- Two level scheduler: events (higher priority) and tasks (lower priority)
- Tasks are atomic with respect to other tasks
- Run-to-completion semantic allows to have single stack for currently running process
- Tasks simulate concurrency, they are asynchronous with respect to events
- Commands and events are not supported to use a lot of time, tasks are used to do computations
- Tasks must never block or spin-wait

TinyOS Scheduling (contd.)

- Tasks can be preempted by events
- Hardware interrupt supported lowest level events
  - Keyword async used if command or event can be called by hardware handler

Task example:

post processData();

```
task void processData() {
  int16_t i, sum=0;
  atomic {
    for (i=0; i < size; i++)
      sum += (rdata[i] >> 7);
  }
  display(sum >> log2size);
}
```

ActiveMessaging

- Abstraction used for message-based communication in parallel and distributed systems
- Assumes that every node runs the same code
- A message consists of the name of a handler called on arrival and data payload as argument
- A typical Handler should perform the following
  - Extract the message from the network
  - Do some local computation
  - Send response if necessary
- Handlers are called on packet reception events and not as tasks, so they should execute quickly

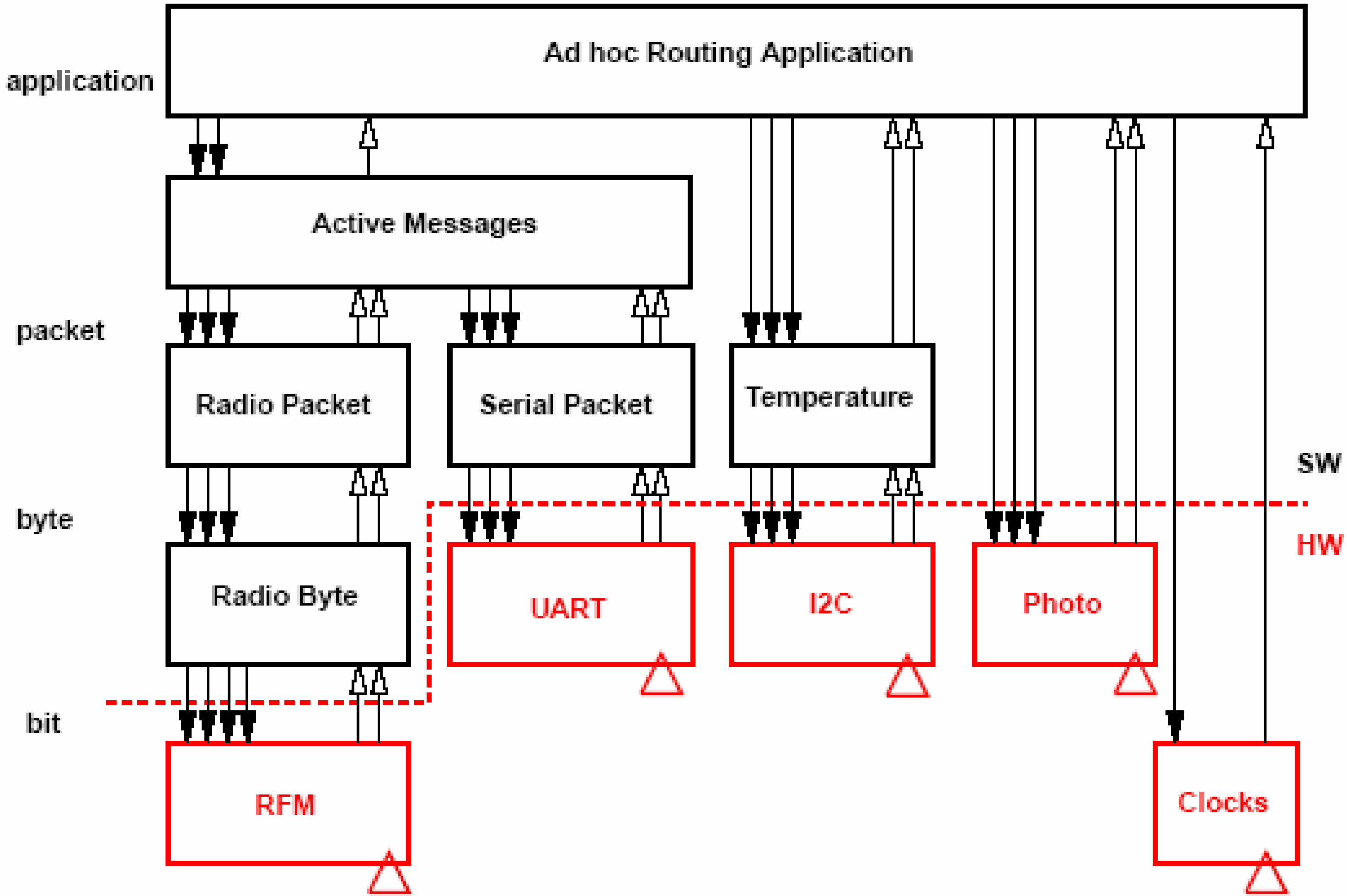
Active Messaging (contd.)

- No need for busy-waiting / receiver buffering
  - Stop and wait semantic not affordable
  - Less memory consumption
  - Pipeline Analogy
- Event centric nature perfectly fits into TinyOS
- Constraints:
  - Active Messaging can only handle one Message at a time
  - Cannot receive while transmitting (half-duplex)
- Three primitives in Tiny Active Messaging:
  - Best effort message transmission
  - Addressing -> Address checking
  - Dispatch -> Handler invocation

Active Messaging (contd.)

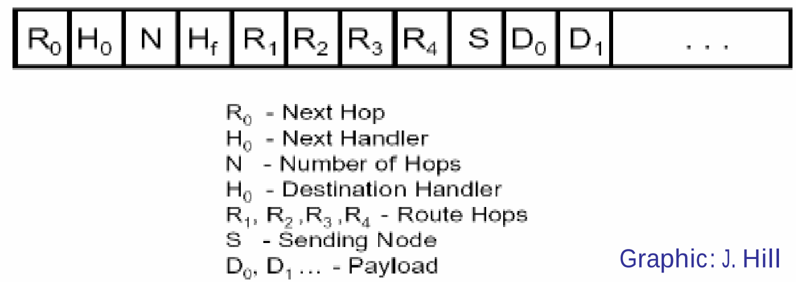
- Modularity: Application chooses between types / levels of error correction / detection
- Application can have additional components
  - Flow control
  - Encryption
  - Packet fragmentation
- Sequence of events
  - Radio bits received by node – RFM
  - Radio bits converted to bytes – RadioBytes
  - Bytes to packets – RadioPacket
  - Packets to Messages – Active Messages

TinyOS Protocol Stack



Graphic: J. Hill

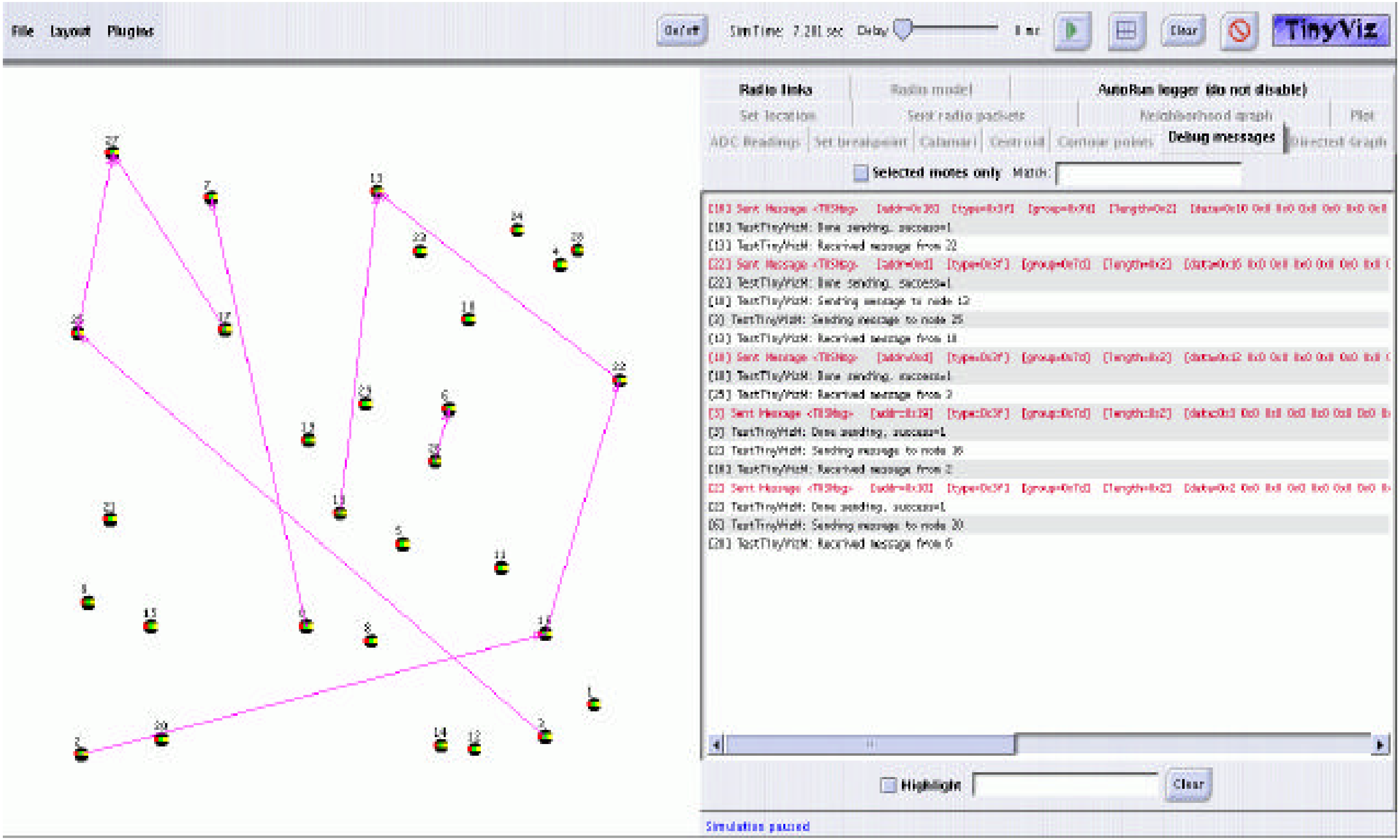
TinyOS MultiHop Routing



Graphic: J. Hill

- Packet format:
  - H0 set to 0
  - 4 hop communication
- At each hop routing handler
  - Decrements hop count
  - Rotates next hop, pushes own address to end
- Route discovery via 2 hop broadcasting followed by self address
  - Returned message contains address of neighbours
- Topology discovered by shortest path from every node to base station
  - Base station broadcasts identity from time to time

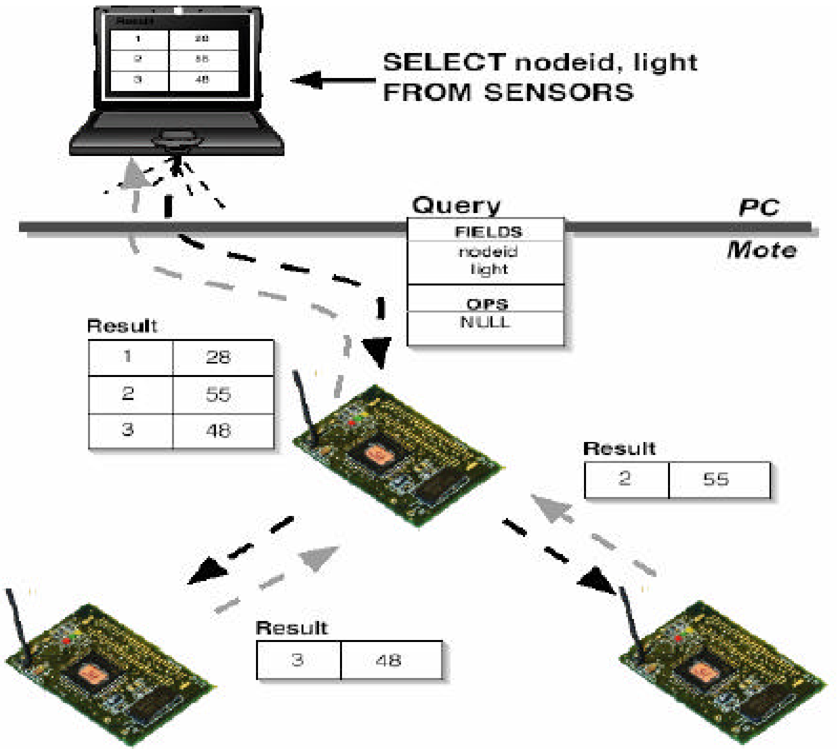
TOSSIM Simulator and TinyWiz



Graphic: G. Wong

TinyDB

- RDBMS-like interface to sensor nodes
- Treat sensors as “virtual table”



Graphic: Arvind Easwaran

TinyDB (contd.)

- Continuous Data Stream

  SELECT nodeid, light, temp FROM sensors SAMPLE INTERVAL 2s FOR 60 s;

- Sorting and symmetric join over stream not allowed (blocking) unless window specified:

  CREATE STORAGE POINT recentlight SIZE 8 AS (SELECT nodeid, light FROM sensors SAMPLE INTERVAL 10s)

  Joins allowed between storage points on same node and between storage point and sensors

- Local triggers allowed

  ON EVENT bird-detect (loc) SELECT AVG(light), AVG(temp), event.loc FROM sensors AS s WHERE dist(s.loc, event.loc) < 10m SAMPLE INTERVAL 2s FOR 30s

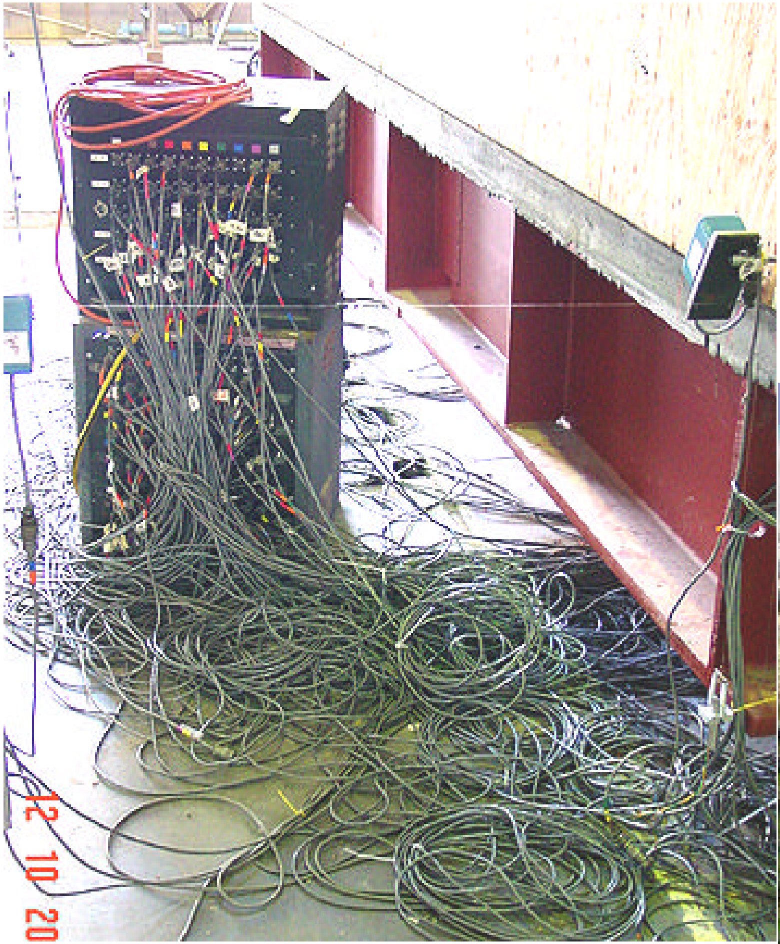
Conclusions

- TinyOS is designed for very small resources
  - Event driven architecture to provide fast transmission of sensor data
  - Some aspects of the architecture (like static resource allocation) are results of the severe resource constraints and will have to be improved in future.
  - Radio Transmission is the major bottleneck, so improve routing
- Hardware
  - Collect data, do some local computation, transmit
  - No open standard for sensor interface but maybe soon, as Intel is pushing sensor network technology
  - Environmental problems: Battery power, but also solar power needs accumulators

References

- A. Easwaran: TinyOS
- M. Franklin, W. Hong: ETH Zürich Distributed Systems Summer School: Data Streams and Sensor Networks
- J. Hill: A Software Architecture Supporting Networked Sensors
- J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister: A System Architecture for Networked Sensors, U.C. Berkeley
- J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister: System Architecture Directions for Networked Sensors, U.C. Berkeley
- V. Raghunathan: TinyOS
- G. Wong: Motes, nesC and TinyOS
- www.tinyos.net: TinyOS Tutorial

TinyOS



That's it.

Thank you for your attention and please ask your questions.