Sensor Network
Platforms and Tools

7.1~7.3 劉立彥
7.4~7.6 許百寬
Introduce

- WSNs are subject to
  - Energy, Bandwidth, Computation, Storage, Real-time constraints, Ad hoc deployment, Frequently changing network topology
- Different from traditional distribution systems
- WSNs can hardly assume an always-on infrastructure.
Introduce

- There are two types of programming for WSNs
  - End users: query to get data from WSNs
  - Application developers: provide end users of a sensor network with the capabilities of data acquisition, processing, and storage.

- This chapter focuses on software design issues.
Introduction

- 7.1 Sensor Node Hardware
- 7.2 Sensor Network Programming Challenges
- 7.3 Node-Level Software Platforms
- 7.4 Node-Level Simulators
- 7.5 State-Centric Programming
- 7.6 Summary
7.1 Sensor Node Hardware

- Sensor node hardware can be grouped into three categories
  - Augmented general-purpose computers
  - Dedicated embedded sensor nodes
  - System-on-chip (SoC)

- Berkley motes due to their small form factor, open source software development, and commercial availability, have gained wide popularity in the sensor network research community.
Augmented general-purpose computers

- Off-the-shelf operating systems such as WinCE, Linux and with standard wireless communication protocols such as 802.11 or Bluetooth.
- Relatively higher processing capability
- More power hungry
- Fully supported popular programming languages
- Ex: PDAs
Dedicated embedded sensor nodes

- In order to keep the program footprint small to accommodate their small memory size, programmers of these platforms are given full access to hardware but barely any operating system support.
- Typically support at least one programming language, such as C.
- Ex: mica, TinyOS, nesC
System-on-chip (SoC)

- Build extremely low power and small footprint sensor nodes that still provide certain sensing, computation, and communication capabilities.
- Currently in the research pipeline with no predefined instruction set, there is no software platform support available.
# Berkley motes

<table>
<thead>
<tr>
<th>Mote type</th>
<th>WeC</th>
<th>Rene</th>
<th>Rene2</th>
<th>Mica</th>
<th>Mica2</th>
<th>Mica2Dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example picture</td>
<td><img src="example.png" alt="Example" /></td>
<td><img src="example.png" alt="Example" /></td>
<td><img src="example.png" alt="Example" /></td>
<td><img src="example.png" alt="Example" /></td>
<td><img src="example.png" alt="Example" /></td>
<td><img src="example.png" alt="Example" /></td>
</tr>
<tr>
<td><strong>MCU</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip</td>
<td>AT90LS8535</td>
<td>ATmega163L</td>
<td>ATmega103L</td>
<td>ATmega128L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>4 MHz, 8 bit</td>
<td>4 MHz, 8 bit</td>
<td>4 MHz, 8 bit</td>
<td>8 MHz, 8 bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program memory (KB)</td>
<td>8</td>
<td>16</td>
<td>128</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAM (KB)</td>
<td>0.5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>External nonvolatile storage</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip</td>
<td>24LC256</td>
<td></td>
<td>AT45DB014B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connection type</td>
<td>I2C</td>
<td></td>
<td>SPI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (KB)</td>
<td>32</td>
<td></td>
<td>ST2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Default power source</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Coin cell</td>
<td>2xAA</td>
<td>Coin cell</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typical capacity (mAh)</td>
<td>575</td>
<td>2850</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RF</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chip</td>
<td>TR1000</td>
<td></td>
<td></td>
<td>CC1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio frequency</td>
<td>868/916MHz</td>
<td></td>
<td></td>
<td>868/916MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw speed (kbps)</td>
<td>10</td>
<td></td>
<td>40</td>
<td>38.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulation type</td>
<td>On/Off key</td>
<td></td>
<td>Amplitude Shift key</td>
<td>Frequency Shift key</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.1** A comparison of Berkeley motes.
Mica mote

- Two-cpu design
  - Main microcontroller (MCU), Atmel
  - Much less capable coprocessor, only active when the MCU is being reprogrammed.
- Memory inside the MCU
  - 512KB flash memory
  - 4KB data memory
- A separate 512KB flash memory unit that can hold data.
- 50kbps raw data rate (40kbps transmission rate)
- 300 feet (90m)
Mica mote
Mica mote

- 51 pin I/O extension connector
- Sensor board
  - Temperature, light, accelerometer, magnetometer, microphone, beeper
- Communicate with a PC in real time through serial I/O (UART)
- Parallel connection is primarily for download programs
Mica mote

- Transmission bears the maximum power consumption
- Energy-saving
  - Suspend the MCU and the RF receiver as long as possible
## Mica mote

<table>
<thead>
<tr>
<th>Component</th>
<th>Rate</th>
<th>Startup time</th>
<th>Current consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCU active</td>
<td>4 MHz</td>
<td>N/A</td>
<td>5.5 mA</td>
</tr>
<tr>
<td>MCU idle</td>
<td>4 MHz</td>
<td>1 μs</td>
<td>1.6 mA</td>
</tr>
<tr>
<td>MCU suspend</td>
<td>32 kHz</td>
<td>4 ms</td>
<td>&lt;20 μA</td>
</tr>
<tr>
<td>Radio transmit</td>
<td>40 kHz</td>
<td>30 ms</td>
<td>12 mA</td>
</tr>
<tr>
<td>Radio receive</td>
<td>40 kHz</td>
<td>30 ms</td>
<td>1.8 mA</td>
</tr>
<tr>
<td>Photoresister</td>
<td>2000 Hz</td>
<td>10 ms</td>
<td>1.235 mA</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>100 Hz</td>
<td>10 ms</td>
<td>5 mA/axis</td>
</tr>
<tr>
<td>Temperature</td>
<td>2 Hz</td>
<td>500 ms</td>
<td>0.150 mA</td>
</tr>
</tbody>
</table>

**Figure 7.3**  Power consumption of MICA motes.
7.2 Sensor Network Programming Challenges

- Event-driven execution allows the system to fall into low-power sleep mode when no interesting events need to be processed.
- At the extreme, embedded operating systems tend to expose more hardware controls to the programmers, who now have to directly face device drivers and scheduling algorithms, and optimize code at the assembly level.
7.3 Node-level software platforms

- Node-centric design methodologies: Programmers think in terms of how a node should behave in the environment.
- A node-level platform can be a node-centric OS, which provides hardware and networking abstractions of a sensor node to programmers.
7.3.1 TinyOS

- No file system
- Static memory allocation: analyzable, reduce memory management overhead
- Only parts of OS are compiled with the application
TinyOS

- A program executed in TinyOS has two contexts, tasks and events.
  - Tasks are posted by components to a task scheduler. Without preemption or being preempted by other tasks
  - Triggered events can be preempted by other events and preempt tasks
TinyOS

- Split-phase operation
  - Command send() → event sendDone()
  - Avoid blocking the entire system
  - Not accepting another packet Until sendDone() is called, avoid race condition
7.3.2 Imperative Language: nesC

- nesC is an extension of C to support the design of TinyOS.
- A component has an interface specification and an implementation.
- A component specification is independent of the component implementation.
- A provides interface is a set of method calls exposed to the upper layers.
- A uses interface is a set of method calls hiding the lower layer components.
module TimerModule {
    provides {
        interface StdControl;
        interface Timer01;
    }
    uses interface Clock as Clk;
}.

interface StdControl {
    command result_t init();
}

interface Timer01 {
    command result_t start(char type, uint32_t interval);
    command result_t stop();
    event result_t timerOfire();
    event result_t timer1Fire();
}

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

Figure 7.7 The interface definition of the Timer component in nesC.

module Timer {
    provides {
        interface StdControl;
        interface Timer01;
    }
    uses interface Clock as Clk;
}

implementation {
    bool evenFlag;

    command result_t StdControl.init() {
        evenFlag = 0;
        return call Clk.setRate(128, 4); // 4 ticks per second
    }

    event result_t Clk_fire() {
        evenFlag = !evenFlag;
        if (evenFlag) {
            signal Timer01.timerOfire();
        } else {
            signal Timer01.timer1Fire();
        }
        return SUCCESS;
    }
...

Figure 7.8 The implementation definition of the Timer component in nesC.
An event call is a method call from a lower layer component to a higher layer component. (signal)

A command is the opposite. (call)

A component may use or provide the same interface multiple times. Give each interface instance a separate name using as notation.
There are two types of components in nesC, depending on how they are implemented: modules and configurations.

- Modules are implemented by application code.
- Configurations are implemented by connecting interfaces of existing components.
  - $A.a = B.a$, the interface $a$ of $A$ is the interface $a$ of $B$
  - $A.a -> B.a$, interface is hidden from upper layers
nesC

Figure 7.9 The TimerC configuration implemented by connecting Timer with HWClock.

Figure 7.10 The implementation definition of the TimerC configuration in nesC.

```plaintext
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer01;
  },
  implementation {
    components TimerModule, Clock;
    StdControl = TimerModule.StdControl;
    Timer = TimerModule.Timer;
    TimerModule.Clock -> HWClock.Clock;
  }
}
```
nesC

- An application must contain the Main module which links the code to the scheduler at runtime.
- The Main has a single StdControl interface, which is the ultimate source of initialization of all components.
nesC—concurrency and atomicity

- A keyword atomic to indicate that the execution of a block of statements should not be preempted.
- Method calls are not allowed in atomic block.
- A shared variable x is outside of an atomic statement is a compile-time error.
- A norace declaration of the variable can prevent the compiler from checking the race condition on that variable.
nesC—concurrency and atomicity

```c
module SenseAndSend()
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;

Implementation {
    bool busy;
    morace uint16_t sensorReading;

    command result_t StdControl.Init() {
        busy = FALSE;
    }

    event result_t Timer.TimerOfFire() {
        localBusy;
        atomic {
            localBusy = busy;
            busy = TRUE;
        }
    }
}
```

```c

task void sendData() { // Send sensorReading.
    adcPacket.data = sensorReading;
    call Send.send(&adcPacket, sizeof adcPacket.data);
    return SUCCESS;
}
```

```c
event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    post sendData();
    atomic {
        busy = FALSE;
    }
    return SUCCESS;
}
```

**Figure 7.11** A section of the implementation of SenseAndSend, illustrating the handling of concurrency in nesC.
7.3.3 Dataflow-style language: TinyGALS

- Dataflow languages are intuitive for expressing computation on interrelated data units by specifying data dependencies among them.
- A data flow program has a set of processing units called actors.
- Actors have ports to receive and produce data.
TinyGALS

Figure 7.13 Implementation of the TimerActor in TinyGALS.

```java
Actor TimerActor {
    include components {
        TimerModule;
        HWClock;
    }
    init {
        TimerModule.init;
    }
    port in {
        timerStart;
    }
    port out {
        zeroFire;
        oneFire;
    }
}

implementation {
    timerStart -> TimerModule.Timer.start;
    TimerModule.Dock -> HWClock.Clock;
    TimerModule.Timer.oneFire -> zeroFire;
    TimerModule.Timer.timerFire -> oneFire;
}
```

Figure 7.14 Triggering, sensing, and sending actors of the FieldMonitor in TinyGALS.

```java
Application FieldMonitor {
    include actors {
        TimerActor;
        SenseAndSend;
        Comm;
    }
    implementation {
        zeroFire => photoSense 5;
        oneFire => TempSense 5;
        send => comm_input 10;
    }
    STARTS timerStart;
}
```

Figure 7.15 Implementation of the FieldMonitor in TinyGALS.

Queue size
7.4~7.6
Node-Level Simulators

- For engineer to perform performance study, which in terms of
  - Power
  - Bandwidth
  - Etc
Node-Level Simulators (2)

- Simulators are consisted by the following models
  - Sensor node model
  - Communication model
  - Physical environment model
  - Statistics and visualization
Time concept

- A sensor network simulator simulates the behavior of sensor network with respect to time
- In which, time may advance in different ways: cycle-driven or discrete-event.
Cycle-driven simulation

- A cycle-driven (CD) discretize the continuous real time into ticks
- Simulator computes phenomenon at each tick. Like: physical environment, sensing data, communication data, etc.
- Communication by RF is assumed to be finished in a tick.
Cycle-driven simulation cont.

- CD simulators are easy to implement and use.
- Most CD simulators issue are detecting and dealing cycle dependencies among nodes (ex: RF) or algorithms (ex: Thread).
Discrete-event simulation

- Discrete-event (DE) simulator assumes the time is continuous.
- Usually use a Global event queue to store events.
- All events are stored chronologically in the Global event queue.
Example figure

- Sending a big file (1MB), 0.1MB/s max.
- CD
- DE
Comparison

- DE simulators are considered as better than CD simulators, because they are more actual. But they’re more complex to design and implement.
- Most popular sensor network simulators are DE simulators, like TOSSIM and NS2.
Ns2 + Sensor network

- Ns2 was meant to be a wired network simulator, so extensions are being made for wireless (802.11, TDMA) and sensor networks.
Ns2 + Sensor network cont.

- Protocol supported:
  - 802.3
  - 802.11
  - TDMA
  - Ad hoc routing
  - Sensor network routing
TOSSIM

- Which is dedicated to TinyOS applications to running on Motes.
- With some visualization packet, the TinyViz. Result can be plot in some more understandable graphs.
State-Centric Programming

- Applications that isn’t just simply generic distributed programs over an ad hoc network. We have to centralize data into nodes.
- EX: target tracking.
State-Centric Programming

- Def:
  - $X$: state of a system
  - $U$: inputs
  - $Y$: outputs
  - $K$: update index
  - $F$: state update function
  - $G$: output observation function
State-Centric Programming

- $X_{k+1} = F(X_k, U_k)$
- $Y_k = G(X_k, U_k)$

- In state-centric programming, $X$ and $K$ come from many nodes. So many issue are discussed.
State-Centric Programming

- Where are the state vars stored?
- Where do the inputs come from?
- Where do the outputs go?
- Where are the functions f and g evaluated?
- How long does the acquisition of inputs take?
Collaboration Group

- Which is a set of entities to update data.
- Protocol example:
  - Geographically constrained group
  - N-hop neighborhood group
  - Publish/Subscribe group
  - Acquaintance group
  - Mixing
Geographically constrained group

- Since some phenomenon will be sensed in an area, GCG is useful.
- By broadcasting from one specific sensor, those who have heard the packet will become the same group.
N-hop neighborhood group

- An anchor sets the hop limit and broadcasting it. Those who heard and is under the limit will become the same group.
  - 0-hop: itself
  - 1-hop: neighbors one hop away
Publish/Subscribe group

- Dynamically defined by the requirement
- Only those have interested data will become the same group.
Acquaintance group

- More dynamically, nodes will be invited to join a group. They can also quit.
- Group leader is selected beforehand, uses an ad hoc routing method to retrieve data from other nodes, then decide which one to invite.
Collaboration Group

- Simulator: PIECES (Programming and Interacting Environment for Collaborative Embedded Systems)
Summary

- Overview of software and hardware of sensor network.
- Most software are tightly bounded with specific hardware. A generic and high performance simulator is expected.
- Programming methodologies are so important to sensor network, like VHDL and Verilog served the VLSI.