nesC

Peter C. Chapin
CIS-3030, Vermont Technical College

Wireless Sensor Networks

- Small, inexpensive nodes (“motes”)
  - Equipped with application specific sensors.
  - Custom software
- Larger base station
  - Could be a laptop
  - Could be a PDA
- Nodes gather environmental data and relay it to the base station.
  - Wireless range limited; multiple hops necessary.

Many Parameters

- Nodes have high failure rate.
  - Network must adapt to lost nodes and paths.
- Reception is variable.
  - Network must adapt to radio fading.
- New nodes might appear at any time.
  - Network must adapt to additional nodes and paths.
- Nodes might move around.
- Various lifetime requirements.

Very Small Systems

- One common theme is that the nodes are all very small.
  - As little as 4K of RAM
  - As little as 16K of program memory
  - Slow processors (1 MHz?)
  - Very low power operation
    - Ideally a node should run for weeks or months on two AA batteries.
    - Must minimize radio communication
  - Very inexpensive
    - Many applications require nodes to be expendable.

Programming Languages

- Assembly Language
  - Not actually used that much.
  - Too low level.
  - Not portable.
- C
  - Commonly used.
  - Easier to program (than assembly), still highly efficient.
- nesC
  - A specialized dialect of C

Component Oriented

- nesC is a “component oriented” language.
  - You define various components (modules) that `provide` and `use` specific interfaces.
  - You compose these components into configurations *after the fact*.
    - Called “wiring” the components.
  - The configurations are also components and can be used in larger configurations.
- Intended to mimic the way electronic components can be wired together.

Example Interface

- This is in a file TimerControl.nc
  -
```
interface TimerControl {
   command error_t setTimeOut(int ms);
   command int getTimeOut();
   command error_t start();
   event   void fired(int count);
}
```
  - Commands are functions you can call in the interface.
  - Events are “call back” functions that you must provide so the interface can call you.

Example Timer Module

- This is in a file TimerC.nc

  ```
  - module TimerC {
        provides interface TimerControl;
    }
    implementation {
        int current_timeout = 0;

        command error_t TimerControl.setTimeOut(int ms)
        {
            current_timeout = ms;
            return SUCCESS;
        }
    }
  ```

- *Must* implement all commands in `TimerControl`.

Example Application Module

- This is in a file MainC.nc
  - ```
    module MainC {
        uses interface TimerControl;
    }
    implementation {
        void f()
        {
            call TimerControl.setTimeOut(250);
            call TimerControl.start():
        }

        event void TimerControl.fired(int count)
        {
            // Do this when the timer fires!
        }
    }
    ```
- *Must* implement all events in `TimerControl`.
    ```

Example Configuration

- This is in a file AppC.nc
  - ```
    configuration AppC {
    }
    implementation {
       components MainC, TimerC;

       MainC.TimerControl -> TimerC.TimerControl;
    }
    ```
- MainC is "wired" to TimerC.
  - TimerControl commands invoked by Main module call into Timer module.
  - TimerControl events invoked by Timer module call into Main module.
    - Neither module is aware of the other.
    ```

Fan-In/Fan-Out

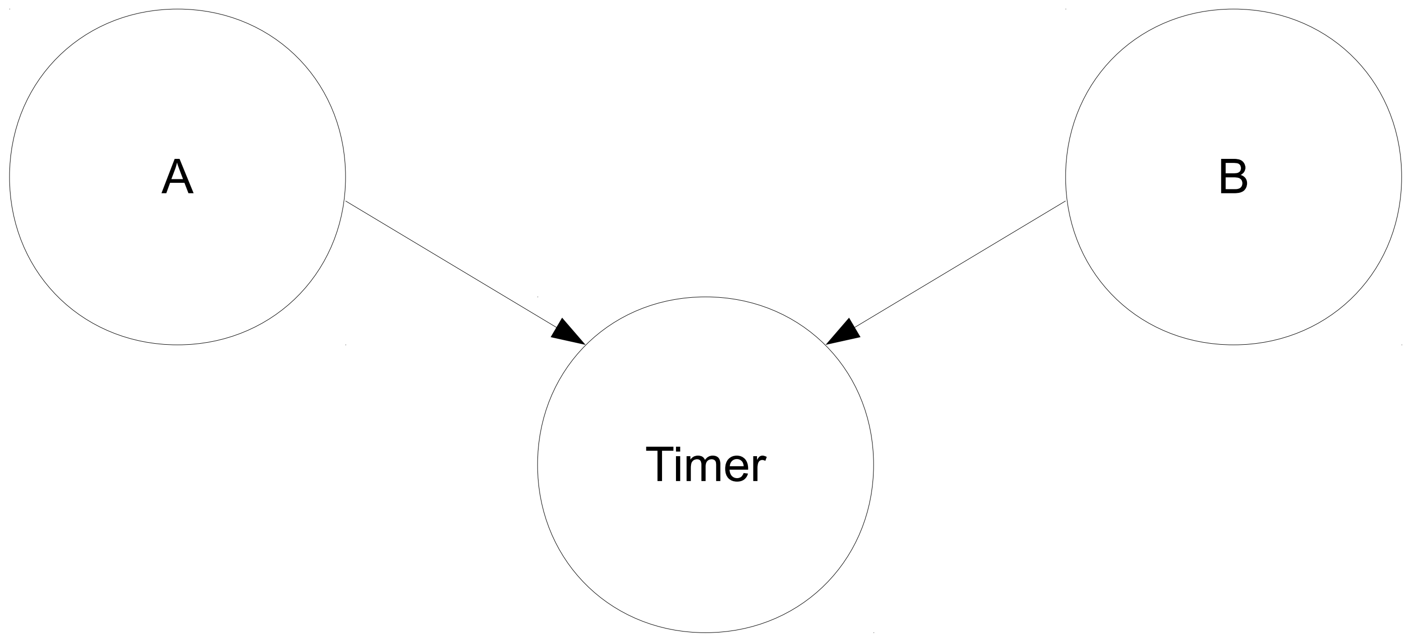
- Consider this

```
– configuration AppC { }
  implementation {
     components A, B, Timer;

     A.TimerControl -> Timer;
     B.TimerControl -> Timer;
  }
```

- TimerControl commands from module A or B invoke code in module Timer. (Fan-In)
- TimerControl events from module Timer invoke code in both modules A and B! (Fan-Out)

Diagram



- When `TimerControl.fire()` is signaled, the implementation in both A and B is invoked.
- Compiler executes them in some order.
- Return values are combined with a *combining function* (user specified, but there are defaults)

Split Phase

- Consider this simple message sending interface
  - ```
    interface SendMessage {
      command error_t send(char *message);
      event void sendDone();
    }
    ```
- To send a message invoke the `send` command.
- The `sendDone` event will be signaled when the message has been sent.
  - Thus the sender does not have to wait for the sending.
  - Can sleep (low power mode) instead.
    ```

Somewhat Bigger Example

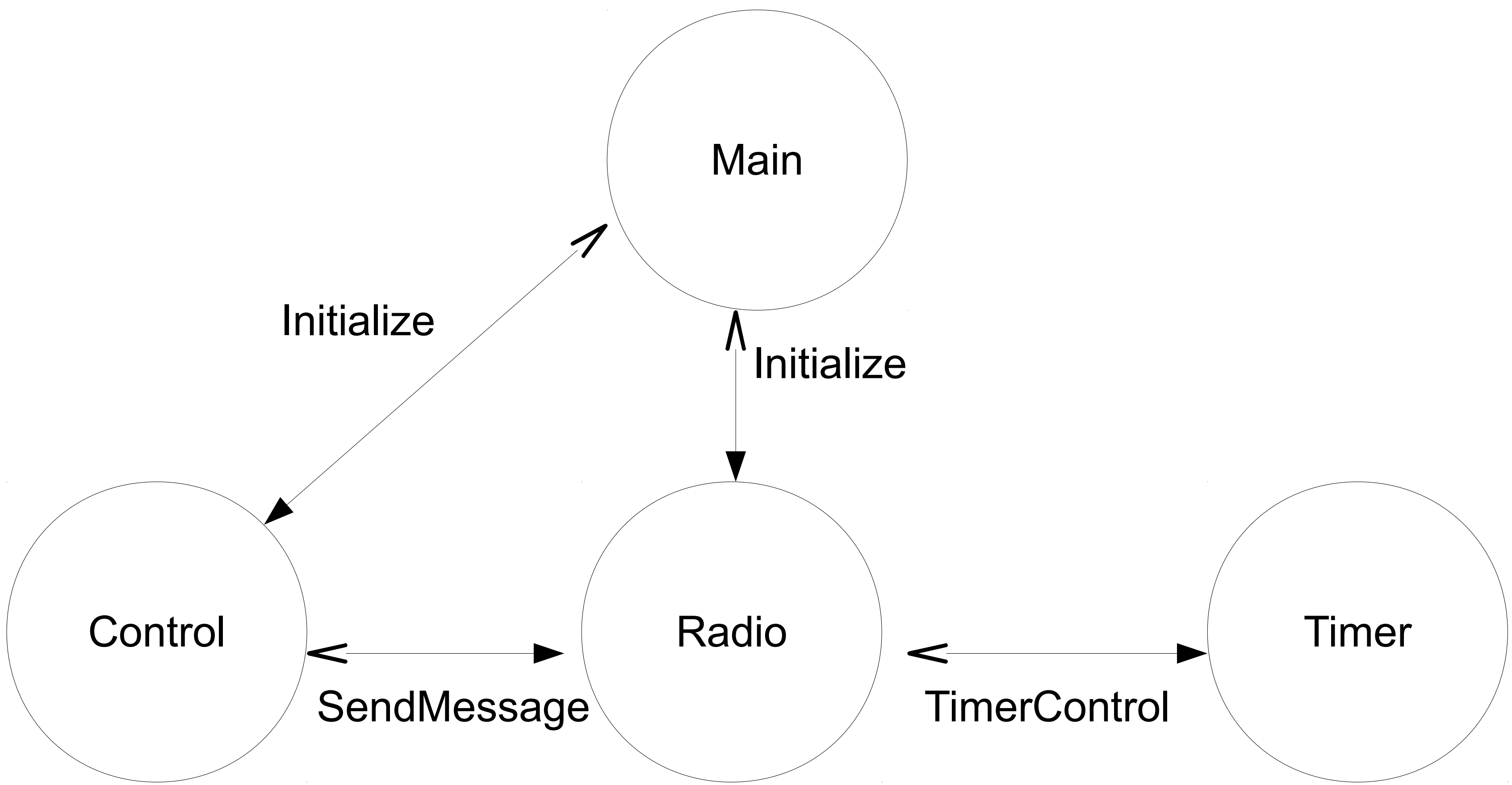
- This is a more complicated module
  - ```
    module RadioC {
       provides interface Initialize;
       provides interface SendMessage;
       uses interface TimerControl;
    }
    implementation {
       // Must implement all commands in
       // Initialize and SendMessage
       //
       // Must implement all events in
       // TimerControl
    }
    ```
    ```

Larger Application

- The main component is always a configuration
  - ```
    configuration AppC { }
    implementation {
      components MainC, ControlC;
      components RadioC, TimerC;

      MainC.Initialize -> RadioC;
      MainC.Initialize -> ControlC;
      RadioC.TimerControl  -> TimerC;
      ControlC.SendMessage -> RadioC;
    }
    ```
    ```

Larger Diagram



nesC Compiler

- The nesC compiler converts nesC to plain C.
  - Reads the entire program at once.
    - Only possible because programs are small
    - Property of sensor network applications
  - Writes a single .c file that is then compiled with a plain C compiler.
- Whole program analysis and optimization feasible.
  - Allows much more efficient code to be generated.
  - C compiler can see entire code base at once.

TinyOS

- An operating system for wireless sensor nodes.
- Written in nesC
  - Shipped as a collection of nesC components.
  - Programmer wires only those components needed
  - nesC compiler builds program from just the components wired.
    - Globally optimizes entire system: application + OS.
    - No components are included that are not used.
- Potentially useful for other embedded systems.

Concurrency in nesC/TinyOS

- Many embedded systems need concurrency.
  - A radio packet might arrive at any time.
  - A timer might say, “time to read the sensors.”
  - A hardware device might generate an interrupt.
- Thread based concurrency is inefficient.
  - Requires that every thread have its own stack.
    - Memory hungry!
  - Requires that “context switching” between threads.
    - Takes too long... especially on a slow processor.

Tasks

- nesC has “tasks” that are “posted”
  - ```
    task do_something()
    {
       // Normal C code.
    }
    ```
- Tasks look like regular C functions inside the implementation of a module.
- Posted with `post do_something();` inside a function, command, event, or another task.
    ```

Run To Completion

- TinyOS has a queue of pending tasks.
  - Each post operation adds to that queue.
- When the node is idle, TinyOS runs tasks from the queue in order.
  - They do not interrupt each other; run to completion
  - A long job might be broken into steps.
    - After each step post another task for the next step.
    - Allows long jobs to be interleaved, but in a simple way.

Interrupt Driven

- A node is driven entirely by hardware interrupts.
  - Sleeps most of the time.
  - When a hardware device (radio, timer, sensor) interrupts...
    - An event is signaled from the module controlling that device.
    - Event handlers execute commands, signal other events, post tasks, etc (directly or indirectly).
  - When the handling of an interrupt is over, the task queue is drained.
  - Repeat!

Split Phase Revisited

- Now we see why split phase is good
  - The `send` command returns quickly.
    - Hardware begins sending.
    - Task queue drains... processor goes to sleep.
  - When the message is sent the hardware interrupts.
    - The radio handling module signals `sendDone`.
    - Application then continues.
    - Sleeps again as soon as possible.

Advantages of nesC Concurrency

- Only a single stack!
  - At any moment there is only a single call stack active. Commands, events, functions, and tasks all use it.
    - Studies have shown that this massively reduces memory requirements.
- No context switching!
  - Only a single thread of execution.
- Simplifies synchronization problems.
  - But doesn't eliminate them. nesC has some additional features in this area.

Take Home Message

- Specialized application domains can benefit from specialized programming languages.
  - Small embedded systems have unusual needs
    - nesC and TinyOS were designed to meet those needs.
- Other special domains
  - HPC (High Performance Computing)
  - Graphics
  - Database
  - etc...
- You may find specialized languages there too.