ActorNet: Actor Language for Wireless Sensor Networks

Slides by: YoungMin Kwon
Kirill Mechitov

Network Embedded Systems

- Low-power, inexpensive embedded processors cannot perform very complex tasks
- But a network of such systems can be very powerful
- Example: sensor networks
  - Each processor is equipped with a sensor
  - Becomes a “smart” sensor node

Wireless Sensor Networks

- Data from multiple sensors is processed and combined into “big picture”
- Sensor coverage
  - Sensors can be deployed to cover a large area
- Reliability
  - Redundant sensor readings
  - Resiliency to failure of individual sensors
- Cost
  - Many inexpensive sensors can be cheaper than one powerful sensor

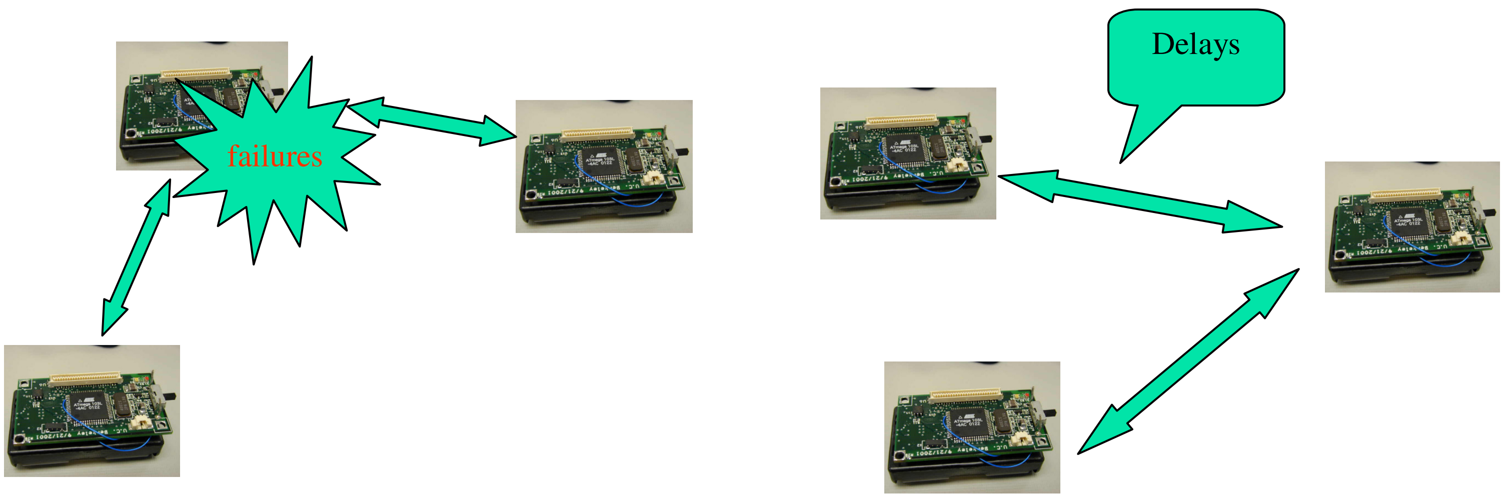
WSN in the lab

WSN in the Field

WSN Environment

Large-scale systems where:

- Nodes and links have limited capabilities.
- Real-time requirements must be met in the absence of a predefined global clock.
- Faults are common.



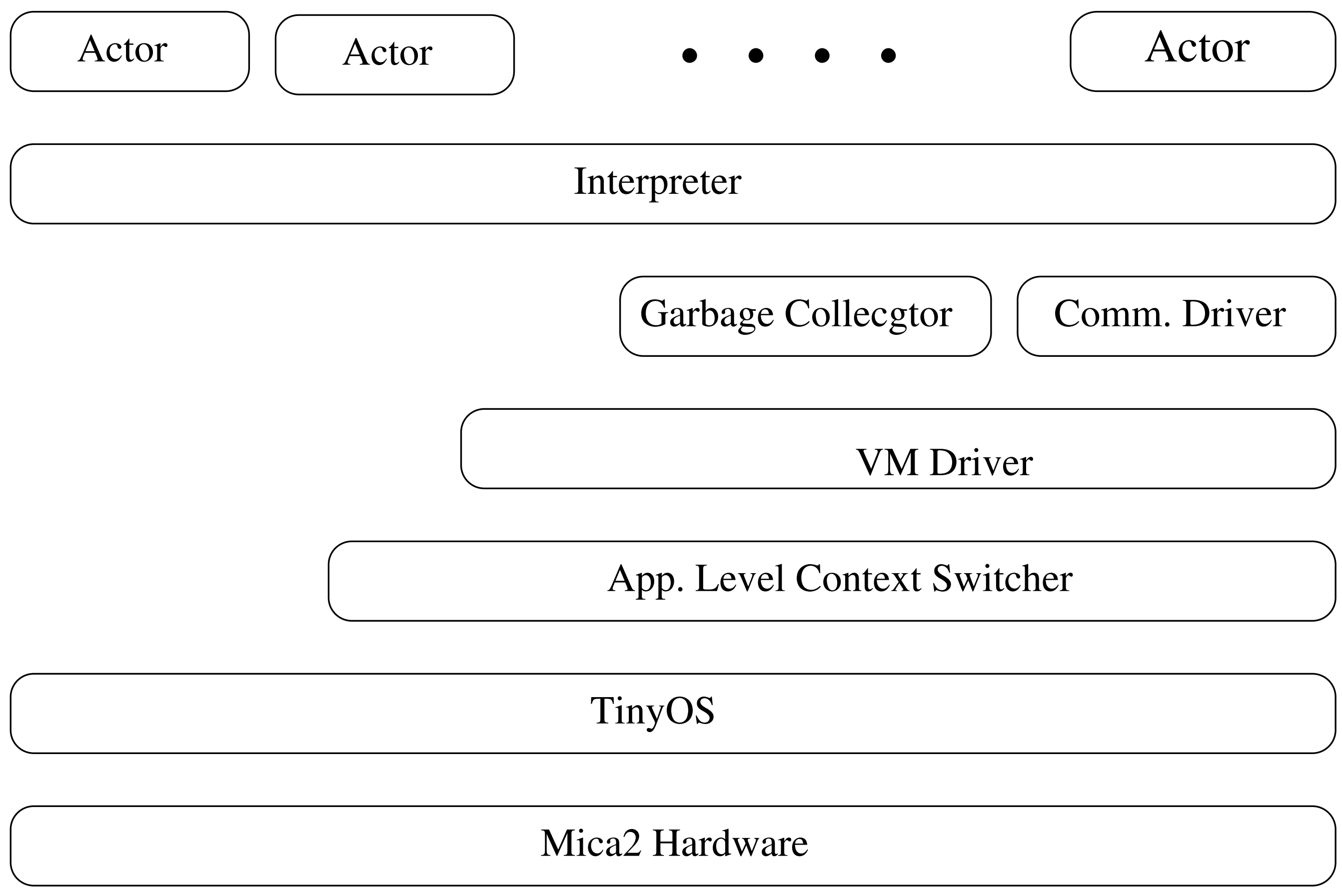
ActorNet

- Easy to program
  - High level language (scheme like)
  - High level operations (e.g. send message)
- Efficient network programming
  - Reprogramming already deployed nodes is very difficult.
  - Deluge: replace every program image in the network
  - ActorNet: migrating actor can run on selected nodes

ActorNet

- Interpreter
  - Provides a uniform computing environment regardless of H/W, O.S. differences.
  - Mica2, PC,…
- Mobility
  - Avoid data collection
  - Efficient way of sampling a sensor network
  - Easily cope with changing requirements on the fly

Software Architecture

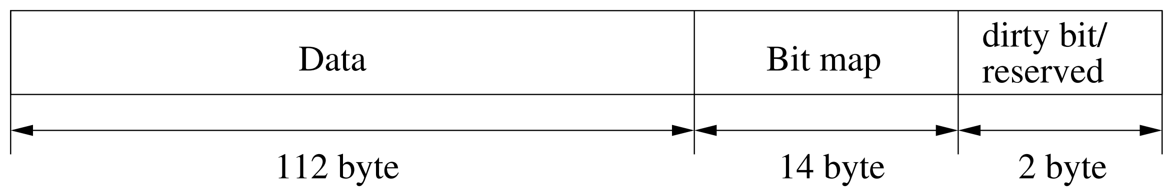


Problems in WSN application development

- Small Memory
  - 4KByte of SRAM
  - 128KByte of program Flash
  - 512KByte of serial Flash (fast read/slow write)
  - All applications as well as TinyOS share the 4KB SRAM

Virtual Memory

- ActorNet provides 56KBytes of virtual memory space
- A page structure



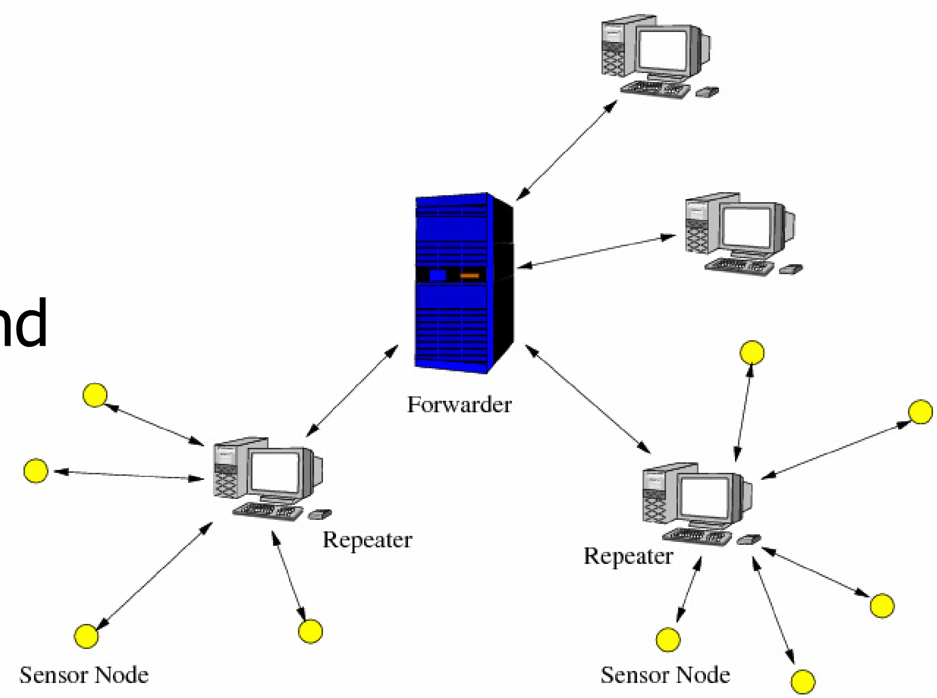
- 1 KByte (8 pages) of SRAM is used as a cache for the VM (LRU swapping policy)
- Lock/Unlock mechanism enables direct memory operation on cached pages

Garbage Collector

- Mark and Sweep garbage collector
  - Mark phase does not take long time if memory is lightly loaded
  - Sweep phase takes long time: it scans entire VM space
- Divide VM into multiple segments
  - Each sweep step scans only one segment
  - Reduce average delay in GC
  - Helps increase the communication speed
    - 0.1 packet/sec -> 2 packet/sec
  - Allocated memory between mark and sweep
    - 2 alternating bit marking
    - New memories are reserved with current mark bit set

Network Structure

- Forwarder
  - Link between repeaters and Actors on PC
  - TCP/IP
- Repeater
  - Link between WSN and the Internet
  - AdHoc network
- An actor can migrate to different network



Interpreter (Scheme like)

- Preorder expression
  - (add 1 2 3) : 6
  - (sub 1 2 3) : -4
- Conditional
  - (cond (ge x 0)
x
(sub 0 x)) : | x |

Function

- Function definition
  - (lambda (x)
    (add x 1)) :increase function
- Function application
  - ((lambda (x)
      (add x 1))
    2) : 3

High-order function

- Let a function DF be

```
(lambda (f)
   (lambda (x)
      (div (sub (f (add x 0.01)) (f x))
           0.01)))
```

- Let fx be (lambda (x) (mul x x))
- Let dfdx be (DF fx)
- (dfdx 5) = 10.01 ~ 10

Recursion

- Summation function: 1+2+…+x
  - ```
    (   (lambda (f)
          (lambda (x)
            (f f x)))
        (lambda (sum x)
          (cond (equal x 1)
            1
            (add x (sum sum (sub x 1))))))
    ```
  - ```
    (rec (sum x)
       (cond (equal x 1)
         1
         (add x (sum (sub x 1)))))
    ```

List structure

- example
  - (cons 1 2) : a pair of 1, 2
  - (car (cons 1 2)) : 1
  - (cdr (cons 1 2)) : 2
  - (cons 1 (cons 2 (cons 3 nil))) ≡ (list 1 2 3)
  - (caddr (list 1 2 3)) ? : 2
- Program is also a list type data
  - (add 1 2 3) ≡ (eval (list add 1 2 3))

Continuation

- Continuation: an abstraction of the rest of the computation
  - (add 1 | 2) :
    - | ≡ (lambda (x) (add x 1))
  - (add 1 | (sub 2 | (mul 3 | 4)) :
    - | ≡ (lambda (x) (add x 1)) ≡ c1
    - | ≡ (lambda (x) (c1 (sub x 2))) ≡ c2
    - | ≡ (lambda (x) (c2 (mul x 3)))

Multi Threading

- A thread's state:
  - a pair of a continuation and a value that will be passed to the continuation
- Multi threading
  - Manages a list of continuation/value pairs
  - Evaluate each pair for a while and switch to the next pair: trampolining
- Each thread (actor) has a unique id and its own message queue

Creating Actors

- (seq (print 1) (print 2))
  - Sequentially evaluates (print 1) and (print 2)
  - Returns 2 which is the value of the last expression.
- (par (print 1) (print 2))
  - Makes two actors that print 1 and 2
  - The expression returns a list of ids of created actors
  - New actor states do not have their parent's continuation stack

Send/Receive Messages

- Message
  - A list that begins with a receiver id
  - 0 for the receiver id means broadcast
- (send (list 100 1 2 3))
  - Send a list of (1 2 3) to actor-100
  - Contents will be deep copied
    - (send (list 100 x)): sends everything reachable from x
- (msgq) returns a list of messages in reverse order
  - (cadr (msgq)) returns the last message
  - (setcdr (msgq) (cddr (msgq))) deletes the last message

Actor migration

- Obtaining an actor's continuation (callcc)
  - ```
    (add 1
            (callcc
                (lambda (cc) (cc 2)))) : 3
    ```
- Actor migration means moving its state (continuation/value pair) to another platform
    ```

Actor migration

- (lambda (adrs val) ;migrate function
  (callcc (lambda (cc)
  (send adrs cc val))))

```
(lambda (adrs val) ;migrate function
   (callcc (lambda (cc)
              (send adrs cc val))))
```

- (add x (migrate 100 y) z)
  - Evaluate x and y
  - Migrate to node 100
  - Evaluate z and add the values of x, y and z at node 100

I/O operations

- (io 0) : hardware ID
- (io 1) : temperature reading
- (io 2) : brightness reading
- (io 3) : clock ticks from the power up
  - 1 tick ~ 0.1 sec