ActorNet: An Actor Platform for Wireless Sensor Networks

YoungMin Kwon, Sameer Sundresh, Kirill Mechitov and Gul Agha *
Department of Computer Science
University of Illinois at Urbana Champaign
Urbana, IL 61801, USA
{ykwon4 | sundresh | mechitov | agha}@cs.uiuc.edu

ABSTRACT

We present *actorNet*, a mobile agent platform for wireless sensor networks (WSNs). WSNs are well-suited to multi-agent systems: agent autonomy reduces the need for communication, saving precious energy. Mobile agents are also an intuitive technique for remotely reprogramming sensors deployed in the field. However, implementing agent programs directly on a WSN is complicated by the many limitations of sensor nodes, including limited memory, slow processors, low bandwidth and finite energy. *ActorNet* eases development by providing an abstract environment for lightweight concurrent object-oriented mobile code on WSNs. As such, it enables a wide range of new dynamic applications, including fully customizable queries and aggregation functions, in-network interactive debugging and high-level concurrent programming on the inherently parallel sensor network platform. Moreover, *actorNet* cleanly integrates all of these features into a fine-tuned, multi-threaded embedded Scheme interpreter that supports compact, maintainable programs—a significant advantage over primitive stack-based virtual machines.

1. INTRODUCTION

A wireless sensor network is a system of autonomous sensor nodes which interact via wireless communication channels. Canonical WSN applications include environment monitoring, target tracking and structural health monitoring [9, 3, 2]. These applications are enabled by the unique features of WSNs: independent energy sources, wireless communication and large-scale deployment of inexpensive devices. Unfortunately, these features also impose severe constraints on WSNs, including limited energy and network bandwidth, small memory sizes and low processing power.

Agent systems are one promising solution to such problems. For example, an autonomous agent can make intelligent decisions on the spot without consulting a central base station, saving energy and bandwidth. Indeed, existing WSN applications such as real-time target tracking already use an agent-based approach [8]. Several challenges become apparent when building agent programs for WSN platforms. Some wireless sensor nodes have only 4KB of SRAM—a severe constraint for even moderately complex applications. Furthermore, sensor node operating systems typically support only non-blocking I/O for performance reasons, complicating application development. The tight coupling between applications and the operating system makes cross-platform interoperability and code migration difficult. In addition, WSN in-network storage services [10], while similar to agent coordination services such as tuple spaces [4], cannot meet their full potential without a supporting agent platform.

ActorNet is an agent platform that addresses these issues, making the development of agent-based solutions to sensing problems significantly easier. Specifically, *actorNet* provides services such as virtual memory, context switching and multi-tasking to support a highly-expressive yet efficient agent language for WSNs. Several agent primitives can be implemented on top of *actorNet*. For example, *actorNet* supports coordination among agents in the asynchronous communication model, typically employed in WSNs due to the gap between communication and computation speeds. Synchronization between concurrent agents in an asynchronous setting can be performed via synchronizers or join continuations, which have previously been defined in an Actor language similar to *actorNet* [1].

2. ACTOR LANGUAGE

The top layer of the *actorNet* platform is an actor language interpreter. The actor language uses the well-known S-expression syntax. Notable primitive operators include `par`, `send`, `msgq` and `callcc`, which are used extensively in this paper.

The *actorNet* language is based on the syntax and semantics of Scheme, with extensions specific to actor computation. The `par`, `send` and `msgq` operators are not found in Scheme. `Par` creates separate actors to evaluate each argument expression in parallel and returns a list of the created actors' identifiers. While these actors remain on the same *actorNet* platform, they share parts of their environments, allowing them to communicate efficiently. If an actor migrate to another platform, it can communicate back via message passing. The `send` operator provides a simple, abstract mechanism to send messages to an actor. A deep copy is made of the message data to prevent any dependence on

*This research has been supported by the DARPA IXO NEST Award F33615-01-C-1907 and the ONR MURI Award N0014-02-1-0715.

the source host. For example, `(send (list 100 x))` sends all data reachable from variable `x` to an actor with id 100. An actor can access its message queue by calling the `msgq` operator, which returns the list of messages the actor has received.

The `callcc` operator accesses the *current continuation* (CC)—an abstraction of the rest of the program remaining to execute. For example, the CC of the expression `(add 1 (mul 2 ↓ 3))` at the ↓ mark can be regarded as a single-parameter function `c1`: `(lambda (x) (c2 (mul 2 x)))`, where `c2` is an another single-parameter function `(lambda (x) (add 1 x))`. In general, the CC can be regarded as a stack of single-parameter functions. The operand of `callcc` is a single-parameter function: when `callcc` is called the CC is passed to the operand function.

In *actorNet*, the CC and the value that will be passed to it form the state of an actor. Because an actor can read its current continuation, it can duplicate itself or migrate to another platform voluntarily by sending its continuation–value pair to another actor. The other actor simply evaluates the continuation on the value to duplicate the sender's behavior. Using these primitives we can easily and intuitively define the agent migration function of Section 3.

```
 1 (rec (move path temp) ;;return path, max temp
 2    (seq
 3       ;;broadcast measure actors to neighbors
 4       (send (list 0 measure (id)))
 5       (delay 100) ;;wait for 10 sec
 6       (  (lambda (maxt)
 7             (par
 8                ;;if it arrives at an maximal point
 9                (cond (le (car maxt) temp)
10                ;;then return the temp along the path
11                   (return migrate path temp)
12                ;;else move to the highest temp. node
13                   (move
14                      (cond (equal path nil)
15                         (cons launch path)
16                         (cons (io 0) path))
17                      (migrate
18                         (cadr maxt)   ;;node id
19                         (car maxt)))) ;;temp
20                (setcdr (msgq) nil)))) ;;reset msgq
21          ;;find the max temp. and the node
22          (max (cdr (msgq)) (list 0 0)))))
```

Figure 1: An example actor program that migrates to a point of maximal temperature in a WSN and returns the temperature back to the base station

3. EXAMPLE

ActorNet is an implementation of the actor model of distributed computing [1] for wireless sensor networks. We present an example application that demonstrates the utility of mobile agents in a sensor network setting. *actorNet*'s support for coordination amongst mobile agents. Our example consists of an actor migrating through the WSN, in search of a local maximal temperature when the temperature is assumed to be continuously changing in space—a typical environment monitoring task. The actor in this example autonomously selects it migration path based on environmental information and reports the final result to a base station.An outline of the application follows.

1. An actor A broadcasts to its neighbors a small actor that measures the temperature at a node and sends back the result.

2. Actor A determines the local maximum temperature and migrates itself to the corresponding node. When it migrates to another node, A records its point of origin so that it can forward the maximum temperature reading back along the path it followed.

3. When it arrives at a point of maximal temperature, A migrates back to the base station. Upon arrival at the base station it prints out the temperature value.

Note that in this example we do not need any support for message routing. An actor locally broadcasts and moves itself to its neighbor with the greatest temperature, and the return path is constructed as it migrates from node to node.

Let us first construct a `migrate` function that makes an actor migrate to another node and continue its execution. The state of an actor can be considered as a pair of a continuation and a value to be passed to the continuation. Using this explicit representation, an actor can easily migrate itself to a neighboring node by sending its current continuation as obtained via `callcc`. Each *actorNet* platform hosts a `launcher` actor which regards messages sent to it as programs and evaluates them. More precisely, `migrate` can be defined as follows:

```
1  (lambda (adrs val) ;; migrate
2     (callcc
3        (lambda (cc)
4           (send (list adrs cc (list quote val))))))
```

Figure 1 shows the code for our temperature-search example. The program first broadcasts a `measure` actor that reads a temperature at a remote node and sends back the reading. The sender then waits for 10 seconds and subsequently checks its message queue, `msgq`, for the measurement. No other work is needed for synchronization. The `measure` actor can be encoded simply as

```
1     (lambda (ret) ;;measure
2        (send (list ret (io 1) (io 0)))).
```

The `(io 1)` call returns a temperature reading and the call `(io 0)` returns the node identifier. The `launcher` actor of a remote platform will evaluate this function with the return address, which is the `id` function call of the 4^{th} line of Figure 1. Although the `measure` actor in this application is simple, it could potentially be any arbitrarily complex function. That is, one can easily distribute complex subprocesses to other nodes and later collect the results in the form of messages. This simple example demonstrates how *actorNet* facilitates multi-agent computation in sensor networks.

Returning to Figure 1, the `move` function takes as arguments a return path and the current maximum temperature reading. Migration occurs after evaluating the second parameter. Line 13 shows how the actor migrates to another node: it first appends its node id—`(io 0)`—to the return path and then migrates to the node where the greatest temperature was read. When it arrives at a point of maximal temperature, it returns the temperature value using the `return` function, listed below.

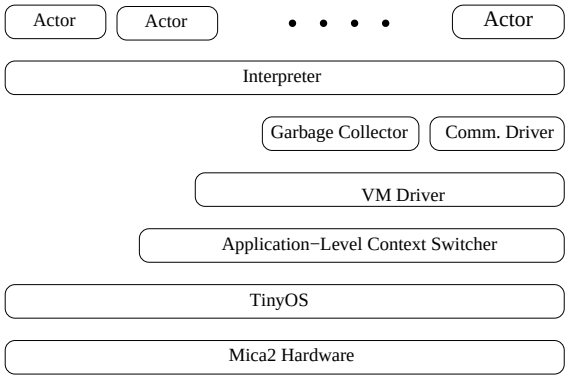

Figure 2: Architecture of the actorNet platform (Mica2 node)

```
1  (rec (return migrate path temp)
2     (cond (equal path nil)
3        (print temp)
4        (return migrate (cdr path)
5                    (migrate (car path) temp))))
```

The `return` function is similar to `move`: it migrates across the nodes along the return path. Incidentally, the `move` and `return` operations can be used to implement a variety of ant-based routing algorithms [6].

4. ACTORNET PLATFORM DESIGN

An *actorNet* is a collection of *actorNet platforms* running on sensor nodes or PCs. Platforms are networked via local wireless channels and the Internet. Each platform hosts several computing elements, called actors, which are able to interact with each other both locally and over the network. Like other virtual machines, the *actorNet* platform provides a uniform computing environment for all actors, regardless of hardware or operating system differences.

Figure 2 depicts the layered architecture of a sensor node *actorNet* platform. Actors only use the interpreter module directly; thus implementation details are hidden from actor programs. Lower-level services are necessary to reconcile the desired properties of simplicity and platform-independence in the high-level language with the specifics of the wireless sensor network environment. The layered architecture alleviates some of the complexity of application development for sensor networks, which currently involves a significant amount of low-level programming due to the tight coupling between the application and the operating system. *ActorNet* aims to provide a stricter decoupling of applications from the operating system, enabling a sensor node to safely load and execute multiple agents, while at the same time simplifying application development. In the same vein, *actorNet*'s support for application-level multitasking makes available such intuitive techniques as blocking I/O, which are usually sacrificed for the sake of efficiency by low-level WSN operating systems such as TinyOS. *ActorNet* maintains this efficiency by introducing threads which are preempted when blocking on I/O.

Several other limitations of present-generation WSN platforms are significant when designing a high-level agent language in this setting, particularly limited memory sizes. We examine representative hardware and software WSN platforms and develop services needed to overcome these limitations.

4.1 WSN Environment

Currently, the sensor node *actorNet* platform is specifically designed for Mica2 hardware. The Mica2 has an 8 MHz 8-bit ATmega 128L CPU with 4 KB of SRAM, 128 KB of program flash memory and 512 KB of serial flash [5]. The 4 KB SRAM space is shared by the stack, heap, and all TinyOS components' static variables. Limited memory heavily constrains applications requiring several coordination services. Application code, large constant tables and logged data are loaded in the flash memory units. ActorNet uses the serial flash as a virtual memory space. Flash read operations are fast, but writes are slow—it takes ∼15ms to write a 128-byte page.

TinyOS is a lightweight operating system for sensor nodes written primarily in NesC [7]. The system is structured as a collection of modules which are statically linked together based on a component specification. Modules consist of statically-allocated variables and three kinds of program blocks: `command`s, `event`s, and `task`s. Service requests are typically split-phase: a caller invokes a command, which returns quickly; once the request is satisfied, the service notifies a corresponding event procedure in the caller. This technique enables higher application throughput as compared to single-threaded blocking I/O. Long-running procedures are explicitly executed as *tasks*, which are scheduled in series and run to completion. Since only interrupts can preempt tasks or lower-priority interrupt handlers, processes must be segmented into sequences of tasks if they are to run concurrently.

4.2 Platform Services

In response to the challenges presented by the WSN environment, we have designed platform services allowing efficient memory management and blocking I/O operations.

Virtual Memory. Since the 4 KB SRAM space on the Mica2 is insufficient for many applications, *actorNet* provides a *virtual memory* (VM) subsystem. We build a page structure on the serial flash area of the Mica2 and use an inverted page table to access pages stored in an LRU cache in SRAM. The current implementation of the VM driver does not perform memory compaction, because the *actorNet* platform does not frequently allocate large chunks of memory; the primary data types are small: bytes, integers, single precision floating point numbers, and pairs of 16-bit addresses (cons cells).

Application-Level Context Switching. The non-blocking command-event I/O model employed by TinyOS is infeasible to support directly in *actorNet*, since every virtual memory access potentially involves split-phase flash memory I/O. For example, let us consider the code in Figure 3. The code on the left assumes blocking I/O: `bar` calls `foo` and `foo` calls `read`, which performs I/O. Without blocking I/O, we must resort to something like the right hand side of Figure 3: the `read` and its callers must be divided into two parts. The problem is that every time `read` is called, the functions along the call chain must be divided into two parts. Furthermore, one cannot use *automatic local variables*: all data must now be declared as static variables outside of the function def-

```
foo() {
   int a;
   ...
   read();
   ...
}
bar() {
   int a;
   ...
   foo();
   ...
}
```

```
int foo_a;
int bar_a;
prefoo() {
   ...
   preread();
}
postfoo() {
   postread();
   ...
}
```

```
prebar() {
   ...
   prefoo();
}
postbar() {
   postfoo();
   ...
}
```

Figure 3: Example program with blocking I/O (left) and without (right)

inition. This makes programs unnecessarily complicated, and also results in inefficient memory usage: local variables must be exclusively allocated as static data rather than as temporary values on a shared stack.

We have devised an application-level context switching mechanism that allows efficient blocking I/O. To explore the utility of the context switching mechanism, consider the following NesC program for `read()`. Note that there is a spin-loop in the `read()` function waiting for `isFlashReadDone` to become true.

```
read() {
   ...
   while(!isFlashReadDone)
      yield();
   return flashData;
}
```

```
task loop() {
   resume();
   post loop();
}
```

With our context switching mechanism, the `yield()` call in `read()` causes control to exit from the `resume()` call of the `loop` task. Thus, TinyOS can schedule other tasks and process pending events. Later, when the `loop` task is scheduled again and `resume()` is called, control continues following the `yield()` call in `read()` as if it had just returned from `yield()`. Note that we do not need to divide the application program into two phases as in Figure 3. Hence the yield–resume mechanism eases development of maintainable applications. To preserve portability and modularity, the context switching mechanism is implemented purely as an application-level service; we do not modify the TinyOS scheduler.

Multi-Phase Garbage Collector. The *actorNet* platform provides a *mark and sweep garbage collection* (GC) mechanism. System-level support for garbage collection has many benefits: it eases application development, eliminates the chance of memory leaks, protects other applications from misbehaving actors, and reduces the actor code size. However, for embedded applications it also has one serious drawback: GC may not always occur at the most opportune moment. This problem becomes especially severe for the *actorNet* platform, because it takes a significant amount of time to write pages to the flash memory. When virtual memory is lightly loaded, the marking step can be done quickly. However, the sweep step must check and restore all pages. In order to solve this problem, we divide the sweep step into many short phases. Combined with the context switching functionality, this greatly reduces the impact of garbage collection on application performance.

5. CONCLUSIONS

We have developed *actorNet*, a mobile agent platform for WSNs. *ActorNet* provides high-level abstractions for coordination in distributed systems, and a set of essential features to develop agent systems on WSNs. Though the current implementation is fully functional, it still has some limitations. One of the biggest remaining challenges is fault tolerence: as message transmission in WSNs is via local broadcast, using a simple message acknowledgement mechanism is infeasible since it requires that each node knows all its neighbor ids'. We are currently investigating techniques to implement a negative-acknowledgement-based rebroadcast mechanism while maintaining as much performance as possible. The virtual memory and multi-tasking environments provided by *actorNet* open the possibility for more advanced coordination mechanisms such as tuple spaces, which can be built on top of existing distributed storage services for sensor networks. Despite the current limitations, *actorNet* provides powerful, efficient high-level services for developing agent systems on WSNs.

6. REFERENCES

[1] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. In *Journal of Functional Programming*, volume 7, pages 1–72. Cambridge University Press, 1997.

[2] A. Basharat, N. Catbas, and M. Shah. A framework for intelligent sensor network with video camera for structural health monitoring of bridges. In *Proceedings of Third IEEE International Conference on Pervasive Computing and Communications (PerCom)*, March 2005.

[3] R. R. Brooks, P. Ramanathan, and A. M. Sayed. Distributed target classification and tracking in sensor networks. In *Proceedings of the IEEE*, 2003.

[4] N. Carriero and D. Gelernter. Linda in context. In *Communications of the ACM*, volume 32, pages 444–458, 1989.

[5] Crossbow Technology, Inc. http://www.xbow.com/.

[6] M. Dorigo, G. D. Caro, and L. Gambardella. Ant algorithms for discrete optimization. In *Artificial Life*, pages 137–172, 1999.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *In Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[8] B. Horling, R. Mailler, J. Shen, R. Vincent, and V. Lesser. Using Autonomy, Organizational Design and Negotiation in a Distributed Sensor Network. In V. Lesser, C. Ortiz, and M. Tambe, editors, *Distributed Sensor Networks: A multiagent perspective*, pages 139–183. Kluwer Academic Publishers, 2003.

[9] A. Mainwaring, J. Polastre, R. S. D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[10] S. Mazumdar. Fast range queries using pre-aggregated in-network storage. Masters' thesis, University of Illinois at Urbana Champaign, 2004.