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

{ykwon4 | sundresh | mechitov | agha}@cs.uiuc.edu

Abstract

We present an *actor platform* for *wireless sensor networks* (WSNs). A typical WSN may consist of hundreds to tens of thousands of tiny nodes embdedded in an environment. Hence, manual reprogramming of nodes for development, fixing bugs and updating features is an arduous process; moreover, in some cases physical access to nodes is simply out of the question. In an attempt to address this problem, network reprogramming tools such as Deluge and MNP [10, 14] have been developed. Unfortunately, these bulk reprogramming services incur significant costs in terms of energy usage, latency, and loss of sensing coverage when nodes are rebooted into a new program image. *ActorNet,* in contrast, provides an environment for lightweight concurrent object-oriented mobile code on WSNs. As such, *actorNet* enables a wide range of new dynamic applications on WSNs, including support for fully customizable queries and aggregation functions, in-network interactive debugging facilities, 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 which supports compact, maintainable programs – a significant advantage over primitive stack-based virtual machines [15, 8].

1 Introduction

A *Wireless Sensor Network* (WSN) is a system of autonomous sensor nodes which interact via wireless communication channels. Canonical WSN applications include environment monitoring [16], target tracking [5], intrusion detection [3] and structural health monitoring [4]. These applications are enabled by the unique features of WSNs, such as independent energy sources, wireless communication channels and large-scale deployment of inexpensive devices. However, these features also impose severe constraints on WSNs, including

a finite energy supply, low network bandwidth, and limited memory and processing power.

Typically, sensor nodes may be deployed in the field, monitor environmental data, and perform some logging or communication. Some applications may require global data metrics, such as the mean and variance of a parameter across the network. One means to maintain a global view is to collect all data at a base station; however, due to limited power, this technique sacrifices network longevity, especially near the base station. On the other hand, if each node is pre-programmed to process data locally, then the system cannot adapt to changing requirements. As such, a mobile agent platform is an attractive solution: agent programs migrate from node to node, and sample and process data on the spot. All data need not be collected centrally, and agents can be injected into the WSN on a need basis. Moreover, it is well known that as the working data set size grows, the overhead incurred due to agent migration becomes negligible; past a certain point, mobile agents prove more cost effective than data transfer.

Communication in WSNs can be different from other network systems. For example, a temperature monitoring application may need to know the maximum temperature across a network. In this case, messages need only be forwarded to the extrema, ideally along the paths of steepest temperature ascent. Generic spanning tree-based packet routing or distance vector routing does not take into account this information. Such environment-dependent routing can easily be implemented with mobile agents. Comparing the amount of data involved in broadcasting commands and collecting data against the cost of agent migration, it becomes clear that for large networks, mobile agents are a plausible, and indeed a desirable technique.

In addition to efficiency, we must consider the level of abstraction in WSN application development. Applications for Mica2 motes are generally developed in NesC. Unfortunately, the low level of abstraction which NesC inherits from C makes it unnecessarily complicated and time-consuming to write code to measure environmental data, exchange data with other nodes, and perform some coordination. To alleviate this development bottleneck, this paper presents a simple actor language that has concise, high-level abstractions for complex operations such as message exchange, multi-threading, and process migration.

The main contributions of this paper are as follows:

1. *Definition of a high-level language for actor programming on wireless sensor nodes.* Our language enables much more expressive, maintainable programs than previous mobile agent systems for wireless sensor networks.

2. *Virtual memory manager with garbage collection for deeply embedded systems.* Our solution overcomes the tight memory constraints of wireless sensor nodes and other microcontroller-based systems, while maintaining good response times.

3. *Application-level multitasking service.* Provides support for multiple threads written to the simple blocking I/O model, while maintaining the efficiency of non-blocking I/O.

4. *ActorNet platform implementations on the PC and Mica2 wireless sensor nodes.*

5. *Evaluation of ActorNet platform performance using a real application.*

2 Related Work

KVM [11] is a compact Java virtual machine designed for embedded systems. As a virtual machine, it provides a uniform computing environment for application programs regardless of hardware and operating system differences. Its class loader [2] makes it possible to dynamically load applications onto nodes via the network, and a form of process migration is possible through object serialization. One of its merits is its applications are written in the popular Java programming language [2]. However, it requires a memory space of 128–256 Kbytes. Hence, it is not suitable for wireless sensor nodes such as the Mica2, which only has 4 Kbytes of SRAM.

Deluge [10] is a network reprogramming protocol especially designed for cases when application code size is larger than node memory space. The protocol works by comparing versions of applications: each node advertises the versions of its applications. If there exists a higher version of the applications on its neighboring nodes, it requests and installs them. With this protocol, we can easily maintain applications on a WSN. However, when we need to update only few nodes, it does too much work. Moreover, some applications are only needed temporarily, and hence need not reside in a node's permanent program memory.

Mate [15] is a high-level stack-based virtual machine for sensor networks. Its high level instructions, like *OPson*, which turns on the sounder, makes application code size small, and make it suitable to transfer code over the network. However, its primitive, assembler-like language makes it difficult to maintain applications.

Agilla is another virtual machine for sensor networks which supports code mobility [8]. Like Mate, Agilla also is a stack-based virtual machine with a powerful instructions set. However, unlike Mate, which supports only a single application program per node, Agilla supports multiple applications. It also supports a Linda-like tuplespace that decouples data from spatial constraints. However, like mate, programmability and code maintainability pose a challenge due to the low level of language abstraction.

Other than the actual implementations of virtual machines, research has also been pursued on the definition of languages for coordination in distributed environments. In [12], rather than sending data, a continuation is transferred to a remote node and data is communicated locally, so as to resolve synchro-

nization issues (communication-passing style). Although we do not send continuations for this purpose, our notion of process migration is similar to this mechanism.

3 Example ActorNet Application

actorNet is an implementation of the actor model of distributed computing [1] for wireless sensor networks. In this section, we present an example application that demonstrates how *actorNet* can be used to create sensor network applications and coordination services. Our example consists of an actor migrating through the WSN, in search of a maximal temperature–a typical environment monitoring task. The example demonstrates the high level of abstraction which *actorNet* provides for WSN application development. An outline of the application follows:

1. An actor A broadcasts to its neghbors 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 where it migrated from so that it can forward the maximum temperature reading back along the path it followed.

3. When it arrives at a maximal temperature point, A migrates back to the base station. When it arrives at a 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. The return path is constructed as it migrates from node to node.

Let us first consider 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 representation, an actor can easily migrate itself to a neighboring node by sending a list containing its current continuation, which can be obtained via `callcc`, and an associated argument value. More precisely, `migrate` can be defined as follows:

```
(lambda (adrs val) ;; migrate
   (callcc
      (lambda (cc)
         (send (list adrs cc (list quote val))))))
```

```
(rec (move path temp)
   (par
      (send (list 0 measure id));;broadcast a measure actor
      (  (lambda (dummy maxt)
            (par
               (cond (le (car maxt) temp) ;;if it arrives at an maximal point
                  (return migrate path temp) ;;then return the temp
                  (move ;;otherwise move to the highest temp. reading node.
                     (cond (equal path nil)
                        (cons launch path) ;;add a launcher actor address
                        (cons (io 0) path)) ;;append its HW-ID to the path
                     (migrate
                        (cadr maxt) ;;migrate to the highest temp. node
                        (car maxt)))) ;;pass the max temp. to the current continuation
               (setcdr msgq nil))) ;;reset msgq
         (delay 100) ;;wait 10 sec
         (max (cdr msgq) (list 0 0))))) ;;find the max temp. and its node
```

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

Note that there is a `launcher` actor running on every *actorNet* platform that regards messages sent to it as the continuation-value list and applies the continuation to the value. As a simple example, the expression `(add 1 (migrate 100 2))` evaluates the numbers 1 and 2 at the current node, migrates itself to node 100, and then adds 1 and 2 at node 100. Note that with this approach, an actor can migrate at any point during its program execution.

Figure 1 shows the code of 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 waits for a 10 seconds and then checks its message queue, `msgq`, for the measurement. The `measure` actor is a simple anonymous function:

```
(lambda (ret)
   (send (list ret (io 1) (io 0)))).
```

The `(io 1)` call returns a temperature reading and the call `(io 0)` returns the node id of the platform. The `launcher` actor of a remote platform will call this function with the return address. Although the `measure` actor is a simple function that reads temperature and returns the reading, it could potentially be any arbitrarily complex function. That is, one can easily distribute complex sub-processes to other nodes and later collect the results in the form of messages. This simple example shows how naturally *actorNet* provides a concurrent programming environment.

The `move` function takes as arguments a return path and the current maximum temperature reading. As we have seen in the simple example of the previous paragraph, migration occurs after evaluating the second parameter. The 9^{th} line of Figure 1 shows how the actor migrates to another node: it first appends its node id– `(io 0)` –to the path and then migrates to the node where the greatest temperature was read. When it arrives at a maximal temperature point, it returns the temperature value using the `return` function, shown below:

```
(rec (return migrate path temp)
   (cond (equal path nil)
      (print temp)
      (return migrate (cdr path) (migrate (car path) temp))))
```

The `return` function performs a similar task as `move`: it migrates across the nodes along the return path. Note how easily one can migrate a process from one node to another. By providing these simple-to-use, high-level features, *actorNet* enables rapid development of powerful WSN applications.

4 Language and Semantics

In this section we describe the syntax and semantics of the *actorNet* language. The syntax and local semantics are similar to those of Scheme [7]. The concurrent semantics at the inter-actor level is defined as a transition relation from one configuration to another.

4.1 Data Structures

The state of an actor is represented as a triple

$$\mathcal{A}(id, continuation, M)$$

where *id* is a unique actor name–represented by an integer, *continuation* is a sequence of actions to be performed by the actor, represented in the form

$$act_1 \rightarrow act_2 \rightarrow \ldots \rightarrow \text{stop}$$

and $M : address \rightarrow value \times value$ is a mutable memory array used to store cons cells. In general, we represent continuations by the variable K. Note that actions in a continuation may include an environment, $E : name \rightarrow value$, which is a mapping from variable names to values. These values may be constants (including actor names) or addresses of cons cells in the memory array, *i.e.,* $value := constant \mid address$. An asynchronous message is represented as a pair

$$\text{msg}(\textit{recipient-id}, \textit{value}).$$

Note that compound values, such as lists, are passed by copying, not as references to another actor's memory space. The configuration of an actor system is represented as a set of actor states and messages

$$\mathcal{A}_1 \mid \cdots \mid \mathcal{A}_m \mid msg_1 \mid \cdots \mid msg_n.$$

4.2 Syntax

The syntax of an actor program is defined as

$$\begin{array}{rcl}
\textbf{Exp} & ::= & \textbf{Num} \mid \textbf{Sym} \\
& \mid & (\texttt{lambda}\ (\textbf{Sym}^*)\ \textbf{Exp}) \\
& \mid & (\texttt{if}\ \textit{test}\text{-}\textbf{Exp}\ \textit{true}\text{-}\textbf{Exp}\ \textit{false}\text{-}\textbf{Exp}) \\
& \mid & (\texttt{begin}\ \textbf{Exp}^*) \\
& \mid & (\texttt{par}\ \textbf{Exp}^*) \\
& \mid & (\texttt{quote}\ \textbf{Exp}) \\
& \mid & (\textbf{Op}\ \textbf{Exp}^*) \\
\textbf{Op} & ::= & \textbf{Prim} \mid \textbf{Exp}
\end{array}$$

where **Num** is the set of numerical values, **Sym** the set of symbols or names, and $\textbf{Prim} = \{\texttt{car}, \texttt{cdr}, \ldots\}$ is the set of primitive procedures built into the *actorNet* platform.

4.3 Semantics

The semantics of actor programs is defined as a transition relation on equivalence classes of configurations. We first specify the equations defining the equivalence classes on configurations. We then define a transition relation $\stackrel{\lambda}{\Rightarrow}$ for the internal transitions on actor states. The interaction amongst actors is defined by a transition relation $\Rightarrow$.

The equivalence class equations on states are as follows:

$$\begin{array}{rcl}
\text{eval}(n, E) \to K & = & \text{val}(n) \to K \\
\text{eval}(s, E) \to K & = & \text{val}(E[s]) \to K \\
\text{eval}(\texttt{(lambda (}\textit{args}\texttt{)}\ \textit{body}\texttt{)}, E) \to K & = & \text{val}(\text{cl}(\textit{args}, \textit{body}, E)) \to K \\
\text{eval}(\texttt{(if}\ \textit{test}\ \textit{texp}\ \textit{fexp}\texttt{)}, E) \to K & = & \text{eval}(\textit{test}, E) \to \text{if}(\textit{texp}, \textit{fexp}, E) \to K \\
\text{eval}(\texttt{(quote}\ \textit{exp}\texttt{)}, E) \to K & = & \text{val}(\textit{exp}, E) \to K \\
\text{eval}(\texttt{(}\textit{op}\ e_1\ \ldots e_n\texttt{)}, E) \to K & = & \text{eval}([\textit{op}, e_1, \ldots, e_n], [\,], E) \to \text{apply}(E) \to K \\
\text{eval}(\texttt{(begin}\ e_1\ \ldots e_n\texttt{)}, E) \to K & = & \text{eval}(e_1, E) \to \text{discard} \to \ldots \to \text{eval}(e_n, E) \to K \\
\text{eval}(\text{exec}(s, K'), E) \to K & = & \text{val}(E[s]) \to K' \\
\text{val}(v) \to \text{eval}([\ldots], [\ldots], E) \to K & = & \text{eval}([\ldots], [\ldots, v], E) \to K
\end{array}$$

The local transition relation $\stackrel{\lambda}{\Rightarrow}$ on actor states is as follows:

$$
\begin{aligned}
\text{val}(\text{true}) \to \text{if}(\mathit{texp}, \mathit{fexp}, E) \to K &\overset{\lambda}{\Rightarrow} \text{eval}(\mathit{texp}, E) \to K \\
\text{val}(\text{false}) \to \text{if}(\mathit{texp}, \mathit{fexp}, E) \to K &\overset{\lambda}{\Rightarrow} \text{eval}(\mathit{fexp}, E) \to K \\
\text{eval}([e, \ldots], [\ldots], E) \to K &\overset{\lambda}{\Rightarrow} \text{eval}(e, E) \to \text{eval}([\ldots], [\ldots], E) \to K \\
\text{eval}([\,], [\ldots], E) \to K &\overset{\lambda}{\Rightarrow} \text{val}([\ldots]) \to K \\
\text{val}([\texttt{\#}\mathit{prim}, ..\mathit{args}..]) \to \text{apply}(E) \to K &\overset{\lambda}{\Rightarrow} \mathit{prim}(..\mathit{args}.., E) \to K \\
\text{cons}(v, v', E) \to K, M &\overset{\lambda}{\Rightarrow} \text{val}(v_c) \to K, M\{v_c \leftarrow (v, v')\} \quad \text{where } v_c \text{ is fresh} \\
\text{car}(v, E) \to K, M &\overset{\lambda}{\Rightarrow} \text{val}(M[v].\text{car}) \to K, M \\
\text{cdr}(v, E) \to K, M &\overset{\lambda}{\Rightarrow} \text{val}(M[v].\text{cdr}) \to K, M \\
\text{setcar}(v, v', E) \to K, M &\overset{\lambda}{\Rightarrow} \text{novalue} \to K, M\{v \leftarrow (v', M[v].\text{cdr})\} \\
\text{setcdr}(v, v', E) \to K, M &\overset{\lambda}{\Rightarrow} \text{novalue} \to K, M\{v \leftarrow (M[v].\text{car}, v')\} \\
\text{callcc}(v, E) \to K &\overset{\lambda}{\Rightarrow} \text{val}([v, \text{cl}([s], \text{exec}(s, K), \emptyset)]) \to \text{apply}(E) \to K \\
\text{val}([v, ..\mathit{args}..]) \to \text{apply}(E) \to K &\overset{\lambda}{\Rightarrow} \text{eval}(v.\text{body}, v.E\{v.\text{args} \leftarrow \mathit{args}\}) \to K
\end{aligned}
$$

Note that the above constitutes the continuation-passing style operational semantics for a basic Scheme-like language. The concurrent transition relation $\Rightarrow$ on actor configurations is as follows:

$$
\begin{aligned}
\mathcal{A}(id, \text{eval}(\texttt{(par } e_1 \ \ldots e_n\texttt{)}, E) \to K, M) &\Rightarrow \mathcal{A}(id'_1, \text{eval}(e_1, E) \to \text{stop}, M) \mid \cdots \mid \\
&\quad \mathcal{A}(id'_{n-1}, \text{eval}(e_{n-1}, E) \to \text{stop}, M) \mid \\
&\quad \mathcal{A}(id, \text{eval}(e_n, E) \to K, M) \\
\mathcal{A}(id, \text{create}(v, E) \to K, M) &\Rightarrow \mathcal{A}(id, \text{val}(id') \to K, M) \mid \mathcal{A}(id', \text{eval}(v, E) \to \text{stop}, [\,], [\,]) \\
&\quad \text{where } id' \text{ is fresh} \\
\mathcal{A}(id, \text{send}(v, v', E) \to K, M) &\Rightarrow \mathcal{A}(id, \text{novalue} \to K, M) \mid \text{msg}(v, \text{marshal}(M, v')) \\
\mathcal{A}(id, \text{recv}(E) \to K, M) \mid \text{msg}(id, m) &\Rightarrow \mathcal{A}(id, \text{val}(v) \to K, \text{unmarshal}(M, v, m)) \quad \text{where } v \text{ is fresh}
\end{aligned}
$$

The `par` and `create` operations allow for implicit and explicit actor creation, while `send` and `recv` enable communication amongst actors. The marshal and unmarshal functions are required to provide a linearized representation of a complex memory graph when compound values are transmitted.

5 ActorNet Architecture

As mentioned, an *actorNet* system is a collection of *actorNet platforms* running on sensor nodes or PCs. Platforms are networked via 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 shows an *actorNet* configuration, consisting of sensor node platforms (small circles) and PC

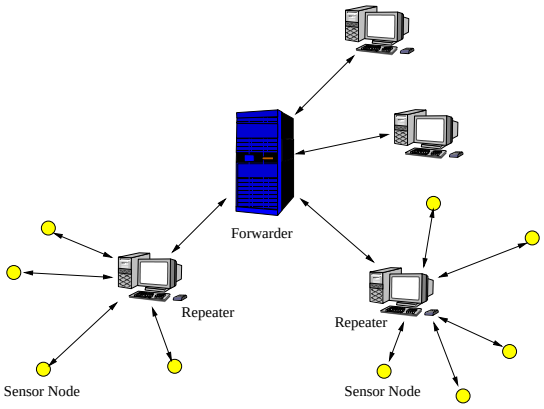

Figure 2: ActorNet architecture

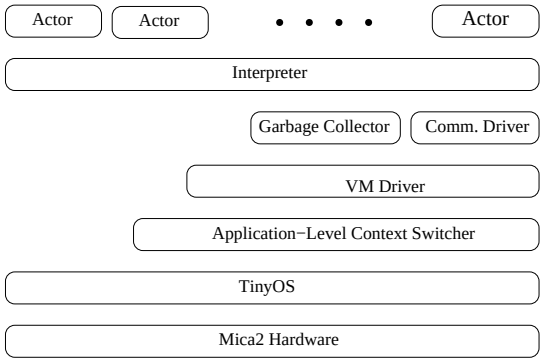

Figure 3: Software architecture of an ActorNet platform (Mica2 node)

platforms. The platforms are connected via two types of server programs called *forwarder* and *repeater*. PC *actorNet* platforms and repeaters are connected through a forwarder via TCP/IP [21]. Repeaters are hosted on PCs, and communicate with sensor networks via a special *GenericBase* [18] sensor node, which serves as a bridge. The forwarder maintains a list of connected repeaters and PC *actorNet* platforms, and broadcasts received packets to all repeaters and PC platforms. Repeaters copy packets from a WSN to the forwarder and vice versa. Because of the bandwidth difference between WSNs and the Internet, repeaters buffer packets from the forwarder to sensor nodes and transmit them gradually.

Figure 3 depicts the layered software architecture of a sensor node *actorNet* platform. Note that while lower layer modules do not make use of higher layer modules, higher layer module may depend on several lower layer modules, not just those directly below. However, actors only use the interpreter module; thus the implementation details are hidden from actor programs.

5.1 TinyOS and Mica2

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 [6]. The 4 KB SRAM space is shared by the stack, heap, and all TinyOS components' static variables. As mentioned earlier, this places a tight memory constraint on applications requiring several coordination services.

Application code, large constant tables, and log data are loaded in the flash memory units. ActorNet uses the serial flash memory as a virtual memory space. Flash memory read operations are fast, but writes are slow–it takes ∼15 ms to write a 128-byte page.

Mica2 hardware is equipped with a CC1000 RF transceiver for single-duplex wireless communication. At the bit-level, TinyOS uses Manchester encoding [21], achieving a theoretical raw throughput of 38.4 Kbits/sec. In practice, we are able to transmit about 20 34-byte packets per second. Internally, TinyOS employs a *carrier sense multiple access* (CSMA) medium access control protocol, called B-MAC [17], together with SEC-DED encoding and a 16-bit *cyclic redundancy code* (CRC) on each packet, which allows recievers to detect data corruption. In addition to the wireless transceiver, Mica2 units feature an RS-232 serial interface [19], allowing communication with PC-based applications through an interface board. A *GenericBase* node is a specially-programmed Mica2 node that serves as a bridge between the wireless and serial channels.

Other notable features of the Mica2 include an independent energy source and a variety of available sensor boards. The ability to run off of a pair of AA batteries allows a Mica2 to be deployed practically anywhere; however, this also imposes strict power usage constraints. Add-on sensor boards included capabilities such as a microphone, photo sensor, magnetometer and accelerometer.

TinyOS is a lightweight operating system for sensor nodes written primarily in NesC [9]. The system is structured as a collection of modules which are statically linked together based on a component specfication. Modules consist of statically-allocated variables and three different types of program blocks: `command`, `event`, and `task`. Service requests are typically split-phase: a caller invokes a command, which returns quickly; once the request is satisfied, the service calls back to a corresponding event procedure in the caller. This communication pattern enables higher application throughput as compared to simple blocking I/O. Long-running procedures are explicitly executed as *tasks*, which are scheduled in series and run to completion. Since only interrupts can pre-empt tasks or lower-priority interrupt handlers, if multiple processes must be run concurrently, they must be explicitly segmented into a sequence of tasks.

5.2 Virtual Memory

Since the 4 KB SRAM space on the Mica2 is insufficient for many applications, *actorNet* provides a *virtual memory* (VM) subsystem. We use 64 KB of the 512 KB serial flash as virtual memory; this space is efficiently indexed by a 16-bit integer. The virtual address space is divided into 512 pages of 128 bytes each. Additionally, 8 pages of SRAM (1 KB) are used as a cache for the virtual memory.

An inverted page table [20] is used to search the cached pages for a requested address. It is implemented as a priority queue that maintains the 8 most recently used pages. Hence, SRAM pages are replaced via an least recently used (LRU) policy. Figure 4 shows the structure of a page. The 128 byte page is divided into a 112-byte data area, a 14-byte bitmap, and a 1-bit dirty bit flag, a 4-bit lock count, and 11 bits of

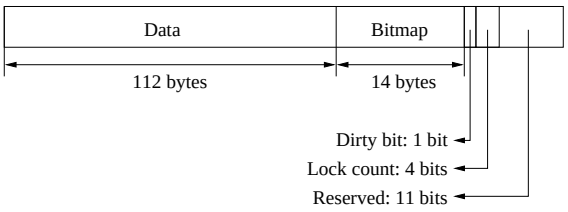

Figure 4: Memory page structure

reserved space. Because flash memory writes are slow, we use the dirty bit to avoid writing pages which are evicted from the cache unmodified. The lock count is used to prevent the VM subsystem from swapping out certain pages. For example, the communication driver of Figure 3 uses 104 bytes of static variables, defined in a structure called `ComData`. Because this data includes buffers which must be shared with the TinyOS communication subsystem, its page must be locked during receive or transmit operations; this is accomplished by calling the VM's `lock` procedure, subsequently followed by a call to `unlock`.

Since there are 112 bytes of data area per page, the effective virtual memory space is 56 KB (512×112). In Figure 4, an allocation bitmap with 8-byte granularity is maintained at the end of each page. Note that the whole 4 KB SRAM space of the Mica2 is not large enough to hold the bitmap of all 56 KB of virtual memory space: $56\text{KB}/8 = 7\text{KB}$. Distributing the bitmap at each page has disadvantages when searching for free space, because a VM driver has to load each page from flash to check for free space. On the other hand, it is crucial to save the precious SRAM space.

The current implementation of the VM driver does not perform memory compaction. This is because the *actorNet* platform does not need to frequently allocate large chunks of memory; the primary data types are small: bytes, integers, single precision floating point numbers, and a pairs of 16-bit addresses (cons cells). The large blocks of memory used in conjunction with the `lock-unlock` mechanism are allocated once when the modules are initialized and never reallocated. As such, fragmentation is a non-issue from a memory availability standpoint.

5.3 Application-Level Context Switching

Most TinyOS I/O operations follow the non-blocking command-event model. While this improves CPU utilization, it also complicates application development. This model would be infeasible to support directly in *actorNet* modules, since every virtual memory access potentially involves split-phase flash memory I/O. For example, let us consider the code in Figure 5. 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 use something like the right hand

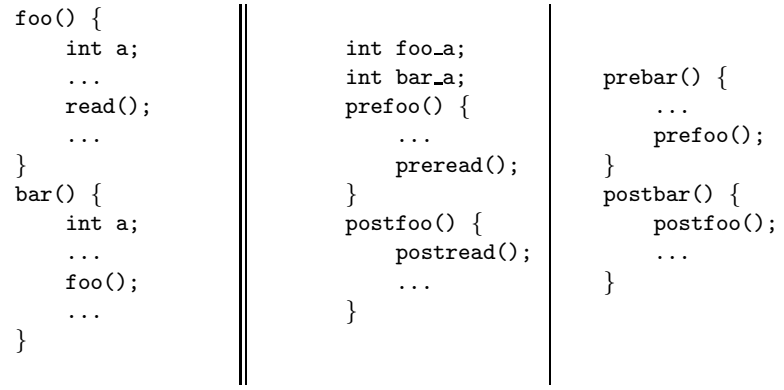

Figure 5: Code examples with and without blocking I/O

side of Figure 5: 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 definition. 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.

In order to address the above problems with non-blocking I/O, we devised an application-level context switching mechanism that simulates blocking I/O. To preserve portability and modularity, this is implemented purely as an application-level service; we do not modify TinyOS itself. In order to explore the utility of the context switching mechanism, let us consider the following NesC program for `read`. Note that there is a spin-loop in the `read` function waiting for the `isFlashReadDone` variable to become true.

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

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

With our context switching mechanism the `yield()` call in the `read` function 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 the `resume` function is called, control continues following the `yield()` call of the `read` function 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 5. Hence the yield-resume mechanism eases development of maintainable applications.

Figure 6 shows pseudocode for the yield-resume mechanism. In order to perform the context switching correctly, stack contents and register values must be preserved. We reserve stack space for TinyOS and other

```
jmp_buf toTos, toApp;
void yield() {
    if(setjmp(toApp)==0)
       longjmp(toTos,1);
}
int resume() {
    int r=setjmp(toTos);
    if(r==0)
       if(/*first time called*/)
           stackBottom(500);
       else
           longjmp(toApp,1);
    else
       return r!=2;
}
void stackBottom(int n) {
    char stack[n];
    /*start ActorNet platform*/
    longjmp(toTos,2);
}
```

Figure 6: Yield-resume: an application level context switching mechanism

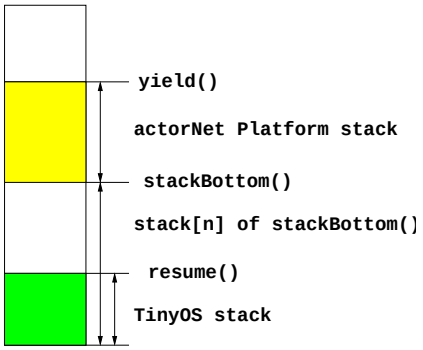

Figure 7: A stack configuration

applications by defining the `stack[n]` array in the `stackBottom` function. Register values including the *program counter* and *stack pointer* are stored and reloaded through the `setjmp` and `longjmp` system calls. The control flow for this mechanism is as follows.

1. When `resume` is called from TinyOS, it stores its register values in `toTos`. If this is the first time `resume` has been called, `stackBottom` is called to allocate TinyOS stack space by defining `stack[n]` array. Following stack reservation, `stackBottom` initiates the *actorNet* platform.

2. When *actorNet* calls `yield`, the current register values are stored in the `toApp` variable and control flow returns from the `setjmp` call of the `resume` function. Note that control does *not* go back to the `stackBottom` function. The value of `r` in `resume` is 1 in this case.

3. When the `resume` function is called again from TinyOS, the register values are restored from the `toApp` variable and control flow is returned to the `setjmp` of the `yield` function.

4. When the *actorNet* platform finally finishes execution, control returns to the `stackBottom` function, and finally back to the `resume` function. Note that the value of `r` is 2 in this case.

Figure 7 shows the stack configuration with this mechanism. In the figure, the stack fills up from the bottom. The shaded area below `resume()` is the stack space used by TinyOS. The white area below

`stackBottom()` is the additional stack space allocated to TinyOS in the `stack[`n`]` local variable. We use $n = 500$ for Mica2 platforms and $n = 5000$ for PC platforms. Note that the TinyOS stack is limited to this white area; while in general, we cannot anticipate stack usage, the applications running on a Mica2 is fixed when a binary image is loaded. This, combined with the fact that most TinyOS applications do not employ recursion, means that in most cases the stack usage is predictable. The shaded area above `stackBottom()` is the stack space used by the *actorNet* platform. The `yield()` line shows the top of the application stack when `yield` is called.

5.4 Multi-Phase Garbage Collector

The *actorNet* platform supports a *mark and sweep garbage collection* (GC) mechanism [13]. System-level support for garbage collection has many benefits: it eases application development, eliminates the chances of memory leaks, protects other actors 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 sweeping step must check and restore all pages. In our experiments, a mark and sweep GC algorithm can take as long as 10 sec. This is a serious problem for communication: because of the memory limit, we cannot prepare enough buffer space for 10 sec while GC is running. We have no choice but to reduce the communication speed: with a buffer for 4 packets we can only reliably send 1 packet every 2.5 sec.

In order to solve this problem, we divide the sweep step into many phases. At each phase a GC routine sweeps 10 pages, which takes about 150 msec. Because *actorNet* has a communication buffer for 4 packets, disregarding the mark phase, ideally, we can send as many as 26 packets per second. However, because we sweep only 10 pages each phase, there are cases when we cannot find sufficient free space in the 10 pages. One important detail in implementing the multi-phase garbage collection algorithm is how to maintain memory allocated after the mark operation but before all sweep phases are finished: if a freshly allocated cell is not marked, it will be swept. One way to solve this problem is to distinguish already-swept pages from not-yet-swept pages and mark freshly-allocated memory only when it is in a not-yet-swept page.

The *actorNet* platform solves this problem in a more efficient way. Instead of using a single bit marking, we use a 2-bit marking: every time the mark function is called, it changes marking bits between 1 and 2. For the fresh memory reserved before sweep phases are done, we mark the memory with the current mark

bit. This prevents the fresh memory from being swept away, and also saves flash write operations by not modifying marked memory to be unmarked.

All *actorNet* platform datatypes have an associated type-tag byte. Since *actorNet* has only 9 types, we can use 2 bits of the tag field for the marking algorithm. Note also that the size of data can be inferred from the the type-tag. As such, the sweep algorithm can efficiently use this information to collect unused memory.

In the *actorNet* platform, actor states are represented as a pair: a continuation and a value which will be passed to the continuation. Thus, a reachability test from this state representation is easy. There is a slight difficulty for an actor that is in the running state, since it may have temporary values that are not yet linked to its state. For these temporary variables, the GC algorithm maintains an array of references to all memory allocated during the execution of an actor prior to its yielding. The GC includes this list in its root set when computing reachability. Note that this reference list is reset every time a new actor is scheduled to execute.

6 Experimental Evaluation

In this section, we evaluate the experimental performance of the *actorNet* platform. First, we examine the page hit ratio of the virtual memory subsystem. Second, we evaluate the performance of the multi-phase garbage collector. Finally, we evaluate the communication costs incurred by *actorNet*.

6.1 Virtual Memory Performance

The current implementation of the *actorNet* platform has 30,960 byte of code and uses 1,967 bytes of data. The code is stored in the Mica2's 128 KB flash memory unit, leaving 100 KB for other applications; the data is allocated in the 4 KB of SRAM space. Because the SRAM must be shared with other applications and TinyOS, it is important to limit the number static variables. As mentioned in Section 5.2, we use the `lock-unlock` mechanism to reliably store and load infrequently used variables from the virtual memory space. The majority of *actorNet*'s memory usage is thus dedicated to the 1 KB virtual memory cache.

We use the following actor program to measure the performance of the virtual memory subsystem and the garbage collector. The program computes the n^{th} Fibonacci number.

```
(rec (fib n)
   (cond (eq n 0)
```

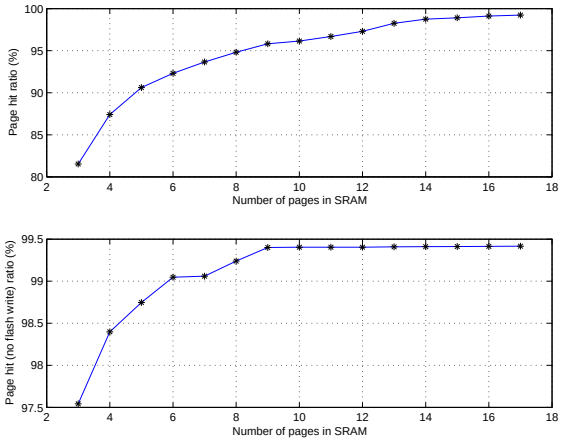

Figure 8: Page hit ratio

```
0
(cond (eq n 1)
   1
   (add  (fib (sub n 1))
         (fib (sub n 2))))))
```

As one might expect, as the page cache size increases, the page hit ratio increases as well. However, in a resource-limited computing environment such as sensor node, we cannot increase the cache size unboundedly: we must consider a tradeoff between performance and the number of applications that can be run on the same platform (since not all applications are written to use our virtual memory manager). The first graph of Figure 8 shows the page hit ratio vs. cache size (the number of pages stored in SRAM). Its shape is approximately concave and increasing with the cache size. After about 14 pages, the slope becomes almost flat. However, in the Mica2 platform, flash write operations dominate the time spent in the virtual memory subsystem. Hence, considering only flash write operations as page-misses is a more accurate performance measure for the *actorNet* platform. The second graph of Figure 8 shows the page hit ratio considering only flash writes as page miss. This graph shows a plateau after 9 cache pages (the current *actorNet* implementation uses 8 cache pages). However, because of the `lock` count, when a message encoding or decoding task is running, it would use 7 cache pages. When there are 8 cache pages, the non-flash-write page hit ratio is 99.24% while with 7 cache pages, the ratio becomes 99.06%.

6.2 Multi-Phase GC Performance

The slow flash write operation of the Mica2 also poses a challenge for the garbage collection service. As mentioned earlier, the GC delay directly limits the communication speed. In order to reduce the delay due

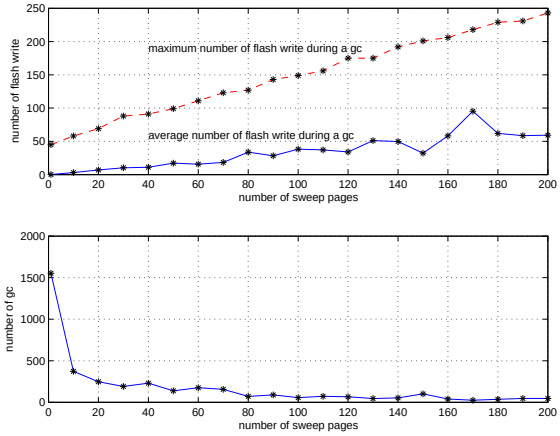

Figure 9: The number of flash writes during a GC phase (top). The number of GC phases called (bottom).

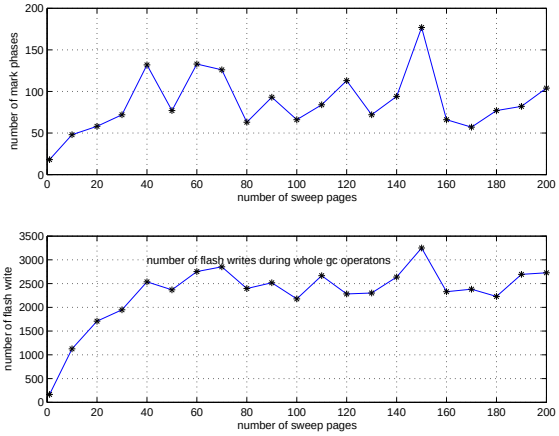

Figure 10: The number of Mark phases called (top). The number of flash writes due to GC algorithm (bottom)

to GC, we have devised a multi-phase GC algorithm. In this section we evaluate the performance of the multi-phase GC algorithm as a function of the number of pages swept per phase. The first graph of Figure 9 shows the number of flash write operations during a GC phase. The solid line shows the average number of flash write operations, which can be interpreted as the expected delay due to GC for each phase, and the dashed line shows the maximum number of flash write operations, which can be interpreted as the worst case GC delay for each phase. The two lines are roughly increasing functions of the number of pages swept, which agrees with our intuition. The second graph of Figure 9 shows the number of times GC is called during an experiment. As is expected, it is a decreasing function of the number of pages swept per phase. The current implementation of *actorNet* sweeps 10 pages per phase; its average number of flash write operations is 3.02 per GC. If we choose the number of pages swept to be 100, then the average number of flash writes is increased to 38.19. That is, when 10 pages are swept per phase, each GC phase takes about 45.3 msec on average, and in the worst case it takes about 870 msec. Currently *actorNet* has a communication buffer for 4 packets; since the repeater of Figure 2 sends 1 packet at every 500 msec, a node can endure upto 2 sec of delay.

There is another merit to multi-phase GC other than the reduced delay of each GC phase. Because our memory reservation algorithm limits the search space for free memory within the interval of the last-swept pages, if the number of pages swept per phase is small, freshly allocated memory addresses are highly correlated in space and time. That is, the fewer the pages swept per phase, the higher the spatial and temporal locality of allocated data. The first graph of Figure 10 shows the number of mark operations

message	content size (byte)	number of messages
`measure`	107	4
`temperature`	27	1
`move`	1629 + hop*8	57+
`return`	474 + hop*4	17+

Figure 11: Amount of message

during an experiment. Note that for each round of GC, there are a single mark phase and multiple sweep phases. Hence, the number of mark phases is an indicator of how efficiently the memory is used. The graph roughly shows that the number of GC rounds increases with the number of pages swept per phase. The second graph of Figure 10 shows the total number of flash write operations made for GC operations during an experiment. It shows an increasing, concave curve: when few pages are swept per phase, related data tends to aggregate. Thus, related data can more likely to be found in cache, which reduces the number of flash write operations. However, sweeping too few pages at a time results in overly frequent call to the GC, as one can see in the second graph of Figure 9.

6.3 Evaluation of Communication Performance

In this section we evaluate the communication costs of *actorNet* applications. We consider the example application of Section 3. As mentioned, this application does not require a routing mechanism. It follows a steepest ascent path of temperatures, and also maintains a return path by itself. Also note that it does not involve spanning tree based data dissemination; the program migrates through the network, rather than collecting all of the data at a central node. When the gradient based path is a straight line, and assuming that nodes are uniformly distributed, the number of nodes involved in this experiment is proportional to $\sqrt{n}$ where n is the number of nodes in the WSN.

Figure 11 shows the number of messages sent in this experiment. Broadcasting a measurement actor to neighboring nodes requires 107 bytes of data in 4 messages. Sending a temperature reading needs 27 bytes, which can be sent in 1 message, if necessary. However, in order to compute the total number of bytes actually sent, this must be multiplied by the number of neighboring nodes. In order to move an actor along the gradient ascent path, 1,629 bytes plus 8 bytes times the hop count thus far are necessary. The hop count-related extra 8 bytes account for the local variables stored during migration (recursion). Note that when the actor migrates back to the base station, it discards unnecessary structure of itself. As such, the returning actor shrinks in size from 1,629+ bytes to 474+ bytes.

7 Conclusions

We have developed an actor platform called *actorNet* for WSNs. *actorNet* provides a high level abstractions for coordination in distributed systems, such as WSNs. The Scheme-like programming environment provides high expressivity features, such as higher-order functions, and the intuitive communication mechanism makes application development easier. The continuation-based actor state representation also makes it clear and easy to manage multi-threading and process migration. *actorNet* also provides a solution for resource-constrained computing environment: its virtual memory and application-level context switching mechanisms enable applications to use large amounts of memory while maintaining code portability. We have demonstrate the utility of the platform via a working temperature monitoring application.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Award No. F33615-01-C-1907 and by the NCASSR Grant N00014-04-1-0562.

References

[1] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. In *Journal of Functional Programming*. Cambridge University Press, January 1993.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 2nd edition, 1998.

[3] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks*, pages 605–634, 2004.

[4] Arslan Basharat, Necati Catbas, and Mubarak 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.

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

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

[7] R. Kent Dybvig. *The Scheme programming language*. Prentice-Hall, 1987.

[8] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Technical Report WUCSE-04-59*. Washington University, Department of Computer Science and Engineering.

[9] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *In Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[10] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[11] J2ME Building Blocks for Mobile Devices. http://java.sun.com/products/cldc/wp/kvmwp.pdf.

[12] Suresh Jagannathan. Continuation-based transformaions for coordination languages. In *Theoretical Computer Science*, volume 240, pages 117–146. Elsevier Science Publishers Ltd., June 2000.

[13] Samuel N. Kamin. *Programming Languages An Interpreter-Based Approach*. Addison Wesley, 1990.

[14] S. S. Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. In *International Conference on Distributed Systems*, June 2005.

[15] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[16] Alan Mainwaring, Joseph Polastre, Robert Szewczykand David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[17] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[18] Projet:TinyOS. http://sourceforge.net/projects/tinyos.

[19] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1992.

[20] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[21] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 4th edition, 2003.