Design and Implementation of a Mobile Actor Platform for Wireless Sensor Networks

YoungMin Kwon[1], Kirill Mechitov[2], and Gul Agha[2]

[1] Microsoft Corporation
One Microsoft Way
Redmond, WA, USA 98052
youngminkwon@gmail.com
[2] Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue
Urbana, IL, USA 61801
{mechitov,agha}@illinois.edu

Abstract. *Wireless sensor networks* (WSNs) promise the ability to monitor physical environments and to facilitate control of *cyber-physical systems*. Because sensors networks can generate large amounts of data, and wireless bandwidth is both limited and energy hungry, local processing becomes necessary to minimize communication. However, for reasons of energy efficiency and production costs, embedded nodes have relatively slow processors and small memories. This makes programming sensor networks harder and requires new tools for distributed computing. We have developed *ActorNet*, an implementation of the *Actor model* of computing for sensor networks which facilitates programming by treating a sensor network as an *open distributed computing platform*. ActorNet provides a high-level actor programming language: users can write dynamic applications for a single cross-platform runtime environment with support for heterogeneous and physically separated WSNs. This shields application developers from some hardware-specific concerns. Moreover, unlike other programming systems for WSNs, ActorNet supports *agent mobility* and *automatic garbage collection*. We describe the ActorNet language and runtime system and how it achieves reasonable performance in a WSN.

1 Introduction

A *Wireless Sensor Network* (WSN) is a system of sensor nodes that collaborate with other nodes through wireless communication channels. A typical sensor node has one or more sensors, some data processing capabilities, a wireless communications channel, and an independent power source. With the utilization of the local processing capabilities and wireless communications, a sensor node can autonomously perform its tasks or collaborate with other nodes. Due to these unique features, WSNs have been proposed for applications such as environmental monitoring [33], structural health monitoring [8], intrusion detection [6], and target tracking [11].

WSN application development remains a complex and challenging endeavor. The task is somewhat simplified by using a distributed middleware which can provide services such as localization [27], time synchronization [34], and data aggregation [40]. Despite the support offered by a middleware, programming WSN still poses difficulties. This is because of several reasons: embedded code is platform dependent; multiple applications cannot be concurrently executed; applications cannot be dynamically loaded; multiple WSNs cannot interoperate; and migration of processes is not supported.

Some of the problems we described above have been studied in the context of *open distributed systems*. In an open distributed system, adding new components may be added, existing components may be replaced, and interconnections between components may be changed. A platform which supports an open distributed largely should allow such evolution without impacting the functioning of the system. The *actor models* provides a suitable for building open distributed systems: it has a notion local components and interaction restricted through specified interfaces. Interaction in the actor model is based on asynchronous message passing; this prevents the direct manipulation of the internal state of one component by another.

In this paper, we implement a variation of an actor model, called *ActorNet* [2] to address some challenges in WSN programming. ActorNet provides a uniform computing platform for mobile agents, which we call *actors* [5]. ActorNet builds a single virtual network by interconnecting physically separated WSNs over the Internet. This virtual network removes the difficulties in interoperating multiple WSNs together. For example, an actor can track a seismic event while migrating thousands of miles through the Internet. The homogeneity of the computing environments provided by the interpreter layer of ActorNet simplifies interoperation. Because the underlying platform differences and the network differences are hidden from the actors, the same actor program can continue its tasks while migrating between different ActorNet platforms, e.g. a Mica2 sensor node and a PC. The ActorNet implementation is available with an open source license[1].

ActorNet consists of a language interpreter and a runtime system. The actor language supports powerful operations such as high-order functions, reflection, garbage collection, and tail recursion removal. The specific details of the underlying hardware and the operating system are hidden behind the high level operators of the actor language. The uniform computing environment also simplifies the application developments greatly as it precludes the need for different variations of programs for different platforms.

ActorNet runtime provides with a library of services such as virtual memory, application level context switching, garbage collection, and a communication stack machine. These services not only secure the necessary resources for ActorNet applications to run, but also enables ActorNet, as an application running on a sensor node, to coexist harmoniously with other native applications.

Unlike other WSN mobile agent frameworks based on a bytecode virtual machine [17], using an interpreter can greater power and flexibility. In particular,

[1] `http://osl.cs.illinois.edu`

the ActorNet interpreter facilitates reflective capabilities of the language. To support actor migration, we uniformly represent the state of an actor as a pair of a continuation [47] and a value to be passed to the continuation. This state representation, along with the reflection capability of the actor language (cf. [53]), endows actors with the *voluntary migration* capability. The mobility of actors enables fine-grained network reprogramming. The actors run only at required nodes and their migration does not disrupt the continuation of other actors' computation.

Organization of the paper. Section 2 discusses the problems we are trying to address. Section 3 describes our approach to addressing the problems identified. Section 4 provides a complete ActorNet example application to illustrate our approach. The detailed syntax and the semantics of ActorNet language is defined in Section 5. Section 6 describes the implementation of the interpreter and the runtime system of ActorNet. In Section 7 we evaluate the performance of the system. Section 8 examines the application of ActorNet mobile agents as the foundation of a *macroprogramming* system. Finally, we discuss the unique contributions of ActorNet in the context of related work on mobile agent systems and network reprogramming in Section 9. Concluding remarks and discussion of future work follow.

2 Motivation

Our research is motivated by our experience in building WSN applications which continues to require embedded systems programming and networking expertise. Domain experts are not usually embedded systems experts, as pointed out in [42]. We believe this has slowed down the adoption of WSNs. The difficulties can be summarized as follows:

– *Embedded code is dependent on the specific platform used.* Thus embedded systems programmers have to be familiar with the intricacies of the hardware, operating system, and programming language used for the particular embedded hardware and software that they are using. Moreover, it is difficult to adapt applications to new sensor platforms, even as new platforms are being continually developed.
– *Interoperation of multiple, possibly heterogeneous WSNs is not supported.* Many large scale events cannot be covered by a single WSN but require multiple WSNs; for example mapping the temperature of a city or recording seismic data observed across the globe may be facilitated by the cooperation of multiple WSNs. However, many WSN applications are designed only for a single or a handful of predetermined groups of sensors.
– *It is difficult to run multiple applications in a WSN.* As we move from dumb sensors to smart sensors with on board processing capabilities, embedded computers will be used to multitask. For example, it may process readings from different sensors and adapt the behavior of these applications based on the readings.

- *Remote reprogramming of sensors is tedious.* Because application images are preloaded on the nodes and the message formats are predetermined, a WSN cannot respond to dynamically changing requirements:
  - an application's coverage is bound to a predetermined set of sensors as nodes;
  - new nodes cannot dynamically join a WSN unless they are already programmed to do so; and,
  - even when applications potentially of interest are known in advance, given that memory on an embedded node is scarce, it is impractical to preload less-frequently used applications on a large number of nodes.

One approach to addressing the problem of dynamically changing requirements is to support *remote reprogramming*. Several network reprogramming systems have been developed including Deluge [22], over-the-air programming of Contiki [15], SOS [20], and Trickle protocol of Mate [31]. These systems install the whole image or replace some of the modules remotely injected from a central node. However, unless remote reprogramming supports fine grained targeting and inter-operation of heterogeneous application images, energy consumption considerations severely limit its usefulness.

A different *remote evaluation* approach has been proposed by Stamos et al. Instead of the traditional client/server architecture, server nodes in this remote evaluation framework provide a set of generic operations which allow remotely transmitted programs to run on a server using the generic operations and return the results [45]. The remote evaluation approach solves the scalability problem and can potentially reduce the communication load as well. However, some tasks can be better executed in a framework that not only allows program to be copied but migrates its state. Migrating an actor's state allows it to continue a computation on the destination platform. Actor migration enables to duplicate a program over the entire network. Moreover, the ability to migrate continuations can reduce the code size that needs to be migrated.

One of the design principles of a WSN is to build a large scale distributed system using cheap, even disposable, hardware. Naturally, problems arising from the limited resources follow. For example, Mica2 [13] node has only 4 kB of memory, which is a very tight limit even for a single application. To make matters worse, TinyOS [50], an operating system for the Mica nodes, does not support dynamic loading and unloading of applications. That is, the 4 kB of memory must be shared by *all* applications shipped on a node. These constraints pose a big impediment to the development and the maintenance of WSN applications. Some embedded computer operating systems support dynamic module/application loading: these include Contiki [15], Mantis [9], and SOS [20]. TinyOS, a popular operating system for WSNs, does not. According to a survey, TinyOS has the largest support community and the largest number of publications among operating systems for WSNs with 81% [29]. For this reason, ActorNet is implemented primarily targeting TinyOS; however, we believe porting ActorNet to other platforms may not be difficult: only a small fraction of the runtime system code is platform-dependent. ActorNet already provides support for two very diverse platforms: TinyOS on Mica2, and Linux on PC.

3 ActorNet Design

We now describe the overall design of the computing environment and highlight the issues that need to be addressed in its implementation. The principal features ActorNet provides are:

– A light-weight actor (mobile agent) programming language for WSN systems which powerful programming constructs such as higher-order functions, reflection, tail recursion removal, and garbage collection.
– Support for multiple concurrently actors which can execute on a node without interfering with each other.
– A library of useful services, including a virtual memory space on embedded nodes to dynamically load and run non-trivial actor programs and an application-level context switching mechanism to enable blocking I/O and fair scheduling.
– A virtual network platform that encompasses multiple physical WSNs and PC platforms without exposing the hardware and networking differences to the application.

3.1 Network Architecture Design

Simplifying the interoperation of multiple physically-distributed WSNs is one of the design goals of ActorNet. Toward this goal, ActorNet builds a single virtual computing environment for mobile actors that encompasses multiple WSNs. Specifically, this environment is constructed by interconnecting the base stations, or gateway nodes, of WSNs via an Internet overlay. Using the virtual environment, differences in the communication network as well as the underlying computing platform can be obscured from application-level actors. Being exposed to these differences, an actor program would have to prepare different sets of handlers for each hardware configuration, which results in duplicated code, unnecessarily complex implementations, and large application code sizes.

The proposed virtual environment spans two tiers of networks: an *ad hoc* wireless network and the Internet, as can be seen in Figure 1. These two network tiers feature vastly different topology, bandwidth, protocols, and performance characteristics. In *ad hoc* wireless networks, messages are locally broadcast to a node's neighbors, whereas most of the Internet consists of wired, point-to-point connections. The bandwidth differences between the two network types can be huge. Typical RF network devices used to interconnect wireless sensors can communicate at speeds ranging from 38.4 to 250 kbps, for 802.15.4 devices. However, in practice the communication speeds are much lower. For example, a 64-node WSN deployed on the Golden Gate Bridge took 12 hours to transport 90 seconds worth of high-frequency vibration data [26]. Finally, there is a multiple order of magnitude difference in performance of the hosts comprising the network: personal computers (PCs) and servers connected to the Internet typically have processors running at several GHz, whereas sensor nodes feature processors with maximum speeds of several MHz. These differences make developing applications that span both network types a challenging endeavor.

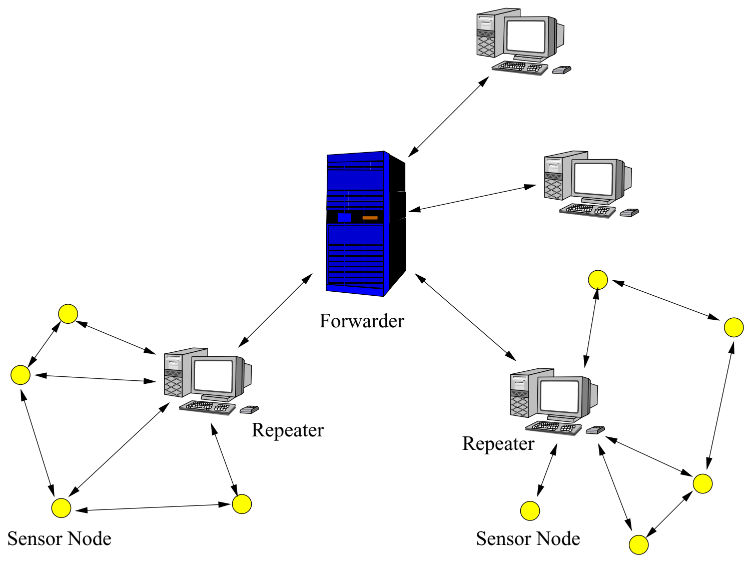


Fig. 1. ActorNet network architecture: the *forwarder* turns the Internet into a single-hop, broadcast overlay network, and the *repeaters* act as a bridge between it and the ad hoc wireless networks. This network architecture obscures the underlying network differences from the actor programs.

To hide the heterogeneity of the network from the actors, we represent the Internet a single hop broadcast network from the actor's viewpoint. Specifically, any messages transmitted from a *gateway node* connected to the Internet are forwarded to the other gateway nodes. In view of this network topology, the difference between the *ad hoc* network and the Internet is hidden from the actors. All ActorNet platforms connected to a forwarder node over the Internet can be regarded as a single hop neighbor. This virtual single-hop network extends the range of mobile actors to the global scale. That means, to an actor, a migration of thousands of miles through the Internet is no different from a local migration between two neighboring sensor nodes. As a solution for the network bandwidth differences, ActorNet provides a packet buffer at the gateway nodes, which compensates for temporary differences in the throughput of the two networks.

One of the merits of this network design is that existing agent coordination algorithms can be easily adopted. For example, *ant algorithms* use a reinforcement based on (computational) pheromones to guide the agent behavior. A notable example is the ant-based routing algorithm [14], which builds robust, adaptive end-to-end routes. In sensor networks, due to the intermittent nature of network connectivity, dynamic routing algorithms based on end-to-end route quality are preferable to static routing tables. With network design, building message routes across the WSNs does not differ from building them within a WSN.

3.2 Actor Language Design

The ActorNet programming language uses the syntax of the programming language Scheme [1], extended with actor operators. In this respect, it is similar to the actor language Rosette [51]. As mentioned earlier, the ActorNet programming language simplifies programming WSNs. This is accomplished by:

Platform independent execution: ActorNet naturally shields the platform differences from actors by functioning as a virtual machine.

Uniform messaging: Communication is platform independent: a simple `send` operator is used to send any type or volume of data to any destination node, even if the destination actor resides in a different WSN connected via an Internet gateway.

Continuation Passing Style (CPS) programming: The state of an actor is represented as a pair consisting of a *continuation*—a single parameter function representing the rest of the program [47], and a *reduction expression* whose *value* is to be passed to the continuation. Applying the value to the continuation produces a new actor state, and an actor repeatedly generates new states as it computes. Continuations allow a programmer to capture, send, and execute future computations, similar to the concept of *future type message passing* in the ABCL/1 concurrent programming language [56].

Actor Mobility: Because an actor program is represented as a data type, it is platform independent and can be migrated as source code.

Concurrency: Multiple actors may be concurrently executed on a single ActorNet node.

Note that the CPS representation of an actor's state supports *reflection* over the current state. This enables actors to migrate themselves at any stage of execution–by accessing the continuation and the value to be passed to the continuation, and send these to a new platform. To make the migration happen, an ActorNet platform needs to be ready to receive an actor's state and let it continue its execution. For this purpose, each ActorNet platform features a special-purpose built-in actor which receives such messages and creates a new actor to evaluate the message content. During the evaluation, the new actor's state is replaced with the actor state in the message.

3.3 ActorNet Platform Design

Running ActorNet platforms on sensor nodes presents its own unique difficulties. In this section, we describe the concerns in developing ActorNet platforms on extremely resource-constrained sensor platforms, such as Crossbow Mica2 sensor nodes (described below). Note that ActorNet would be much easier to implement on more powerful sensor platforms; we use a the Mica2 to demonstrate that our approach can be supported on a broad range of WSN devices.

Mica2 and TinyOS

The Crossbow Mica2 mote is built on an 8 MHz 8-bit ATmega128L CPU with 4 kB of SRAM, 128 kB of program flash memory, and 512 kB of serial flash [13]. The 4 kB SRAM space is shared by the stack, heap, and static variables of all TinyOS components and applications. This, in turn, places a tight memory constraint on applications. Application code, large constant tables, and log data are loaded in the flash memory units. As an application, ActorNet also has to share this 4 kB space, but because its data is actor programs, the small memory is the more restrictive to ActorNet compared to other applications. To address this fundamental problem, we designed a 56 kB virtual memory formed at the 512 kB serial flash memory. Usually, flash memory read operations are fast, but write operations are slow and expensive in terms of energy consumption. On Mica2 it takes ~15 ms to write a 128-byte page to flash.

Mica2 hardware is equipped with a CC1000 RF transceiver for single-duplex wireless communication. At the bit-level, TinyOS uses Manchester encoding [48], achieving a theoretical raw throughput of 38.4 kbps. In practice, a Mica2 node is able to transmit approximately 20 34-byte packets per second. Internally, TinyOS employs a *carrier sense multiple access* (CSMA) medium access control protocol called B-MAC [41], together with SEC-DED encoding and a 16-bit *cyclic redundancy code* (CRC) on each packet, which allows receivers to detect data corruption. In addition to the wireless transceiver, Mica2 units feature an RS-232 serial interface [46], allowing communication with PC-based applications through an interface board.

TinyOS is a lightweight operating system for the sensor nodes written primarily in NesC [19]. The system is structured as a collection of modules which are statically linked together based on a component specification. The 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 a 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 preempt tasks or lower-priority interrupt handlers, if multiple processes must be run concurrently, they have to be explicitly segmented into a sequence of tasks.

In order to enable the mobility of actors, ActorNet supports dynamic loading and unloading of actor programs. Because an actor may allocate and deallocate memory during its computation, the dynamic unloading module must reclaim all the dangling memory made by the unloaded actor. As a general solution to this problem, we added a *Garbage Collection* (GC) mechanism to ActorNet. The GC mechanism is based on the mark and sweep GC algorithm, which is effective but induces an unpredictable latency with a large mean and a large variance. While the GC is running, most of the services are stopped, which can be critical for periodic sampling or communication services. The large variance in the latency also prevents an efficient task scheduling. As a remedy to these limitations, we

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

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

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

Fig. 2. Code examples with (left) and without (right) blocking I/O. `read` makes an I/O operation.

developed a multi-phase GC algorithm. Assuming that the virtual memory is lightly loaded and thus the mark phase is fast, we divided the sweep phase into multiple sub-phases and deallocated only fractions of the memory on each step. This partial deallocation reduces the slow flash memory write while maintaining the page hit ratio high. The multiple phase GC algorithm also reduces the mean and the variances of the GC latency, which results in a better scheduling.

Note that this garbage collection is constrained to local node resources, since the overhead costs involved with implementing distributed GC in a resource-constrained sensor network are prohibitive. In particular, garbage collection of actors is complicated by the fact that not only references from reachable actors have to be considered, but inverse references from potentially active actors must also be considered [52].

TinyOS achieves concurrency among applications through split-phase programming. More specifically, most of the I/O operations are supported only in the split-phase style. Although this style of programming increases throughput, the limited support of the blocking I/O makes it difficult to develop and maintain applications. For example, let us consider the code in Figure 2. The code on the left side is written with blocking I/O: `bar` calls `foo` and `foo` calls `read` which performs an I/O. Without blocking I/O, we must split the functions as in the right side of Figure 2: an application calls `prebar` and arranges an *I/O completed* event handler to call `postbar`. The problem is that every possible function call chain reachable to `read` should be divided into two parts like Figure 2. Furthermore one cannot use *stack allocated local variables* across the divided functions. That is, all such variables must be declared as static variables which take up space even after the functions are returned. The problem is even graver in ActorNet: any memory access can make a page fault which leads to a flash memory access, an I/O operation; split phasing on every memory access is practically impossible. To overcome these problems, we implement an *application level context switching mechanism.* The mechanism enables ActorNet to return control to TinyOS and regain control later with the same register, flags, and stack configurations. Using this mechanism, ActorNet enables blocking I/O for its actors. This mechanism also provides a seamless concurrent execution of ActorNet alongside other applications. The context switching mechanism is developed at the application layer to reduce the difficulty of porting it to other platforms.

The state representation of an actor can be structurally complicated. Specifically, there can be loops or multiple references in the list structure. Thus, sending and receiving the list data involves serialization and deserialization. One of the concerns in sending a message is that the message is locally broadcast to all the neighbors of the sender. Because there is a single sender with multiple receivers, it is computationally beneficial to allow higher computational load at the sender in order to lower the computational load at a receiver. To achieve this, we design a simple *communication stack machine*: a sender handles all the complexities of communication and makes a stream of data mixed with stack manipulation commands so that receivers can restore the data simply by running their stack machine following the commands.

4 Example

We provide a complete example application to illustrate the design of the ActorNet platform before going into the details of the actor language and the platform implementation.

Consider an actor migrating through a WSN in search of a local temperature maximum—a typical environmental monitoring task. Searching for a local maximum point is a reasonable monitoring task for a WSN: for example, to detect a heat sink or a gas leak, one may want to find the local maximum point. An actor in this example autonomously selects its migration path based on the environmental information and reports the final result to a base station. The example demonstrates the high level of abstraction for WSN application development provided by ActorNet.

The steps in our *maxima search actor's* execution are as follows:

1. An actor A broadcasts to its neighbors a simple actor which measures the temperature at a node and sends back the result.
2. A determines the neighborhood 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 with maximal temperature (i.e., where all the neighbors report a lower temperature), A migrates back to the base station. Upon arrival at the base station it prints out the temperature value. (Note that more information could easily be maintained and reported).

Because of expressiveness of ActorNet, we do not need any platform or OS support for multi-hop message routing. An actor locally broadcasts and moves itself to its neighbor with the greatest temperature, while constructing the return path as it migrates from node to node.

```
(rec (move path temp) ;;return path, max temp
 (seq
   (send (list 0 measure (io 0))) ;;broadcast measure actors to neighbors
   (delay 100) ;;wait for 10 seconds
   ( (lambda (maxt)
       (par
         (cond (le (car maxt) temp)    ;;if it arrives at a maximal point
           (return migrate path temp) ;;then return the temp along the path
           (move                      ;;else move to the highest temp. node
             (cond (equal path nil)
               (cons launch path)
               (cons (io 0) path))
             (migrate
               (cadr maxt)   ;;node id
               (car maxt)))) ;;temp
         (setcdr (msgq) nil))) ;;reset msgq
     (max (cdr (msgq)) (list 0 0))))) ;;find the max temp. and the node
```

Fig. 3. An actor program that migrates to a point of maximal temperature in a WSN and returns the temperature back to the base station

Consider a `migrate` function which makes an actor move to another node and then continue its execution. Recall that the state of an actor is a pair of a continuation and a value to be passed to the continuation. In ActorNet, an actor can easily migrate itself to a neighboring node, using the explicit state representation and sending its *current continuation.* There is a `launcher` actor running on every ActorNet platform that receives the messages sent to it as programs and evaluates them. The entire actor migration process is implemented using this very short `migrate` function:

```
(lambda (address value) ;; migrate
  (callcc
    (lambda (cc)
      (send (list address cc (list quote value))))))
```

The code for the temperature-search example, which utilizes this migrate function, is listed in Figure 3. The precise syntax and semantics of the language will be described in the next section, but this code excerpt illustrates the general structure of an ActorNet program and the compactness of the actor language. Note that a relatively complex application is implemented in under 20 lines of code.

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 then checks its message queue, `msgq`, for the measurement. No other work is needed for synchronization. The `measure` actor can be encoded simply as

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

The (io 1) system call returns a temperature reading and the call (io 0) returns the unique node identifier. A launcher actor running at a remote platform will evaluate this function with the return address, which is the (io 0) function call of the 3rd line of Figure 3. Although the measure actor in this example is simple, it could be an arbitrarily complex function. That is, an actor can easily distribute a complex piece of its code to run in other nodes and later collect the results in the form of messages. This example demonstrates the versatility of ActorNet as a concurrent computing environment for multiple actors.

Returning to Figure 3, the move function takes a return path and the current maximum temperature reading as its parameters. Migration occurs after evaluating the second parameter. Line 9 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 the actor arrives at a point of the maximal temperature, it returns the temperature value using the return function, listed below.

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

The return function is similar to move. It migrates across the nodes along the return path.

Note how easy it is to write a mobile agent program using the ActorNet platform. By providing simple-to-use and high-level features, ActorNet enables a rapid development of powerful WSN applications. Furthermore, because mobile agents operate autonomously, they can be used in resource-constrained sensor networks that do not provide many supporting services.

Finally, it is worthwhile to mention that this program does not require any routing services: the actor follows a steepest temperature ascent path, and it also maintains a return path by itself. Also note that the application does not require collecting the temperature reading from all nodes to a central node (usually done by a data dissemination process); instead the actor collects and processes the information while migrating in a sensor field. Even considering the data aggregation service, the saving in the amount of communication by the mobile agent approach is very significant.

5 Actor Language

We now formally describe the syntax and the semantics of the ActorNet actor language in *rewriting logic* [35, 38]. One of the merits of using rewriting logic is that it describes both the syntax and the semantics of a language together. The syntax is defined by its mix-fix definition of operators and the semantics is described by the deductions rules of a rewriting theory. Another benefit of using rewriting logic is that the descriptions are executable by rewriting engines, such as Maude [35].

5.1 Rewriting Theory

In rewriting logic, a *signature* Σ comprises a set S of *sorts*, a partial order relation $\leq$ of *subsorts*, and a $S^* \times S$ indexed set of *operators*. A *Σ-algebra* A_Σ is an algebra with an S indexed family of sets $\{A_s : s \in S\}$ such that $A_s \subseteq A_{s'}$ if $s \leq s'$, and constants $c \in A_s$ for each operator $c_{\emptyset \times s}$ of Σ, and functions $f : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ for each operator $f_{s_1,\ldots,s_n \times s}$ of Σ. An interesting Σ-algebra is the *term algebra* T_Σ whose terms are $c \in A_s$ for $c_{\emptyset \times s}$, and $f(t_1, \ldots, t_n) \in A_s$ for $f_{s_1,\ldots,s_n \times s}$, where t_i is a term in A_{s_i}. The term algebra is a minimal Σ-algebra that has Σ-homomorphism to all Σ-algebras. The mix-fix operator definition of Maude eases defining the syntax of a language. However, for simplicity, we use the BNF notation where possible. For example, we write $\mathcal{P} ::= \langle\, \mathcal{V}\, ,\, \mathcal{V}\, \rangle$ instead of $\langle\ \_\ ,\ \_\ \rangle : \mathcal{V} \times \mathcal{V} \to \mathcal{P}$ for pairs.

An *equational theory* is a pair (Σ, E) of a signature Σ and a set E of possibly conditional equations on the terms of T_Σ. We say that a Σ-algebra A_Σ is a *model* of a theory (Σ, E), and write $A_\Sigma \models (\Sigma, E)$ if A_Σ satisfies all equations in E. An equation e is a *theorem* of (Σ, E) if all models of (Σ, E) satisfy e. Theorems can be proved by applying the deduction rules of *reflexivity*, *symmetry*, *transitivity*, *congruence*, and *modus ponens*. Theorems can also be simply proved by applying *equational rewriting* under the termination and confluence conditions. The equational theory can be generalized in *membership equational logic*, where a *kind* is given to the equivalent class of sorts related by $\leq$, and the operators are indexed with kinds. Sorts are given to the terms through the *membership axiom*.

A *rewriting theory* is a four-tuple $\mathbb{R} = (\Sigma, E, L, R)$, where (Σ, E) is an equational theory and R is a set of labeled rewrite rules whose labels are from L. $\mathbb{R}$ describes the behaviors of a transition system, where the equivalent classes of terms represent the state of the system and the state transitions are described by applying the inference rules of *reflexivity*, *transitivity*, *congruence*, and *replacement*. A more detailed discussion of rewriting logic is presented in [35].

5.2 Syntax

In our actor language, everything is a value sort: numbers, symbols, pairs, lists[2], and all ActorNet program elements, such as actor programs, actor states, and actor configurations are values. Because these program elements have distinguishable structures, specific sorts are assigned to them through membership axiom.

Actor language has only one kind that all sorts belong to. Thus, in this paper, we drop the index from the sorts. Some examples of the sorts in S are $\mathcal{V}$ for values, which is the supersort of the other sorts, $\mathcal{N}$ for numbers, $\mathcal{S}$ for symbols, $\mathcal{P}$ for pairs, $\mathcal{L}$ for lists, $\mathcal{E}$ for expressions, $\mathcal{R}$ for environments, $\mathcal{A}$ for actor states, $\mathcal{K}$ for continuations, $\mathcal{M}$ for actor messages, and $\mathcal{C}$ for actor configurations. For each

[2] Lists are nested form of pairs ending with an empty list. However, because lists simplify the descriptions, we gave them a separate sort.

non-string sort T, we assume that there is a sort $T* \in S$ for the string of the sort. For example, $\mathcal{N}* \in S$ is a string of numbers. In this paper, we denote the variables of a sort with a small letter of that sort. We also suffixed the variables with s for their string sorts.

Examples of the constant operators for symbols ($\emptyset \to \mathcal{S}$) are `lambda, rec, cond, nil`. In this paper, we use the typewriter style font for the symbols. The constructor for pairs is $\mathcal{P} ::= \langle\, \mathcal{V}\,,\, \mathcal{V}\,\rangle$ and the constructor for lists is $\mathcal{L} ::= (\mathcal{V}*)$. A list indeed is a nested form of pairs. Thus, we equate them so that these terms belong to the same equivalent class.

$$\langle\, v\,,\, (vs)\,\rangle = (v\ vs).$$

The expressions $\mathcal{E}$ of the actor language have the well-known *S-expression* syntax defined as follows.

$$\begin{aligned}\mathcal{E} ::= {} & \mathcal{N} \mid \mathcal{S} \\ \mid\ & (\texttt{lambda}\ (\mathcal{S}*)\ \mathcal{E}_{body}) \\ \mid\ & (\texttt{rec}\ (\mathcal{S}\ \mathcal{S}*)\ \mathcal{E}_{body}) \\ \mid\ & (\texttt{cond}\ \mathcal{E}_{test}\ \mathcal{E}_{true}\ \mathcal{E}_{false}) \\ \mid\ & (\texttt{quote}\ \mathcal{V}) \\ \mid\ & (\mathcal{E}_{op}\ \mathcal{E}*),\end{aligned}$$

where $\mathcal{E}_{body}, \mathcal{E}_{test}, \mathcal{E}_{true}, \mathcal{E}_{false}, \mathcal{E}_{op}$ are expressions. When a value term is structured as above, it is given with a sort $\mathcal{E}$.

5.3 Semantics

In this section we describe the semantics of the actor language. First, we explain an informal semantics with examples, and then we describe the formal semantics of the actor system with a rewriting theory $\mathbb{R}$. In $\mathbb{R}$, the transitions of actor states and actor configurations are described as the deductions on the congruent terms modulo equations E.

Informal Semantics. Like the programming language Scheme, the ActorNet language uses prefix notation. For example `(add 1 2 3)` returns 6. Actor language has arithmetic operators like `add, sub, mul, div`, and logical operators like `and, or, not`. It also has a set of pair and list manipulation operators. For example `(cons 1 2)` returns a pair of 1 and 2, and `(car (cons 1 2))` returns the first element 1, and `(cdr (cons 1 2))` returns the second element 2. `(list 1 2 3)` returns a list $(1, 2, 3)$ which is equivalent to `(cons 1 (cons 2 (cons 3 nil)))`. Note that `(cdr (list 1 2 3))` is $(2, 3)$. There are assignment operators `setcar` and `setcdr` that set the first and the second elements of a pair.

An expression beginning with `lambda` is an anonymous function definition, where $\mathcal{S}*$ are zero or more names for the function parameters. To ease writing recursive functions, the actor language has the `rec` primitive, where $\mathcal{S}$ is for the name of the function and $\mathcal{S}*$ is for the parameters. `cond` is used for branching expression: if $\mathcal{E}_{test}$ is evaluated to be true, $\mathcal{E}_{true}$ is evaluated; otherwise $\mathcal{E}_{false}$ is evaluated. Observe that this behavior is not the call by value semantics of the function application. `quote` is an operator that returns its parameter as a value without evaluating it. This operator is useful when we are sending a list as a data to another ActorNet platform. Without this operator, building a literal list is difficult because the interpreter regards the literal list as a function application and tries to evaluate all the elements. The `seq` operator is similar to the `begin` operator of the programming language Scheme: each expression is evaluated in turn, and the value of the last expression is returned.

The `par,` `send` and `msgq` are new actor operators not in Scheme. `par` creates new actors for each expression and makes the actors evaluate the expressions in parallel. The return value of the `par` expression is a list of the created actor ids. Note that these ids are initially known only to the creator, but they can be sent to other actors for the actor coordination, such as the join continuation [5]. While these actors remain in the same ActorNet platform, they share some parts of their environments so that they can communicate efficiently. If actors migrate to another platform, they can communicate via asynchronous messages. The `send` operator provides a simple mechanism to send messages to an actor. `send` makes a deep copy of the message and transmits it to the receiver to prevent any dependence on the source host. For example, `(send (list 100 x))` sends all the data reachable from the 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 the messages the actor has received. ActorNet internally uses a `recv` method that receives the massage and collects it to the list returned by `msgq`. Note that `msgq` is one of the operators that makes our Actor language non-functional; it may return different values for different calls.

The `callcc` operator accesses the *Current Continuation* (CC)–an abstraction of the rest of the program remaining to execute [24]. 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 3)))`, 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 to which the CC is passed.

In ActorNet, the state of an actor is a pair of a CC and a value to be passed to it. 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 ActorNet platform. By simply applying the continuation to the value, the sender's computation is continued on a new platform. Using these primitives we could easily and intuitively define the `migrate` function of Section 4.

Formal Semantics. The state of an actor is a pair of a continuation and a value to be passed to it. In the rewriting theory, the actor states are represented by an equivalent class of terms corresponding to the intermediate computations between the actor state transitions. The computation of an actor is a sequence of actor state transitions. The interaction between actors are captured by an actor configuration which is a snapshot of the whole actor systems. The actor configuration can be regarded as a soup of actor states and actor messages. The concurrent computations of actors are the transitions of the actor configurations.

Let us begin the formal semantics of Actor language with the *environment* ($\mathcal{R}$) which maps the identifiers to their values. An environment comprises two stacks of symbols and values. When a symbol is evaluated, it is looked up from the stack of symbols and the value at the corresponding index in the value stack is returned. $\mathcal{R}$ has three operations: a constant *emptyEnv*: $\emptyset \rightarrow \mathcal{R}$, an extend operation $[\_,\_]/\_ : \mathcal{S}{*}\times\mathcal{V}{*}\times\mathcal{R} \rightarrow \mathcal{R}$, and a lookup operation $\_[\_] : \mathcal{R}\times\mathcal{S} \rightarrow \mathcal{V}$. The equations below describe how environments are built. By the equations, the terms in the left side and the right side of = are put to the same equivalent class of terms[3]

$$\begin{aligned}\text{emptyEnv} &= (\texttt{env}\ ()\ ())\\ [ss', vs']\,/(\texttt{env}\ (ss)\ (vs)) &= (\texttt{env}\ (ss'\ ss)\ (vs'\ vs)).\end{aligned}$$

We also assign the sort $\mathcal{R}$ to the lists structured as $(\texttt{env}\ (\mathcal{S}{*})\ (\mathcal{V}{*}))$. The look up operation is also explained by the following equations.

$$\begin{aligned}(\texttt{env}\ (s\ ss)\ (v\ vs))[s] &= v\\ (\texttt{env}\ (s'\ ss)\ (v\ vs))[s] &= (\texttt{env}\ (ss)\ (vs))[s] \text{ if } s \neq s'.\end{aligned}$$

The second equation is a conditional equation: the equation is applied if the (in)equality following the *if* keyword holds.

Continuations ($\mathcal{K}$) are single parameter functions that represent the rest of the program. In the rewriting logic, we assign a sort $\mathcal{K}$ to the lists structured as follows.

$$\begin{aligned}\mathcal{K} ::= {} & (\texttt{halt})\\ \mid {} & (\texttt{app}\ (\mathcal{V}{*}_{yet})\ (\mathcal{V}{*}_{done})\ \mathcal{R}\ \mathcal{K})\\ \mid {} & (\texttt{if}\ \mathcal{E}_{true}\ \mathcal{E}_{false}\ \mathcal{R}\ \mathcal{K}),\end{aligned}$$

where $\mathcal{V}{*}_{yet}$ is a list of not yet evaluated parameters, $\mathcal{V}{*}_{done}$ is a list of already evaluated parameters, and $\mathcal{E}_{true}/\mathcal{E}_{false}$ are expressions to be evaluated when T/F are passed respectively.

A *state of an actor* is a pair of a continuation and a value: $\mathcal{A} ::= \langle\, \mathcal{K}\,,\, \mathcal{V}\,\rangle$. Any pair structured as such is assigned with a sort $\mathcal{A}$. An *actor configuration* ($\mathcal{C}$) is a set of actor states and actor messages. *Actor messages* is a list structured as $\mathcal{M} ::= (\texttt{mesg}\ \mathcal{N}\ \mathcal{V})$, where $\mathcal{N}$ is the recipient address and $\mathcal{V}$ is the message contents. The sort $\mathcal{C}$ of actor configurations is a supersort of $\mathcal{A}$ and $\mathcal{M}$, and has an associative and commutative constructor: $\_\,|\,\_ : \mathcal{C}\times\mathcal{C} \rightarrow \mathcal{C}$. That is, $\mathcal{C}$ is a soup of actor messages and actor states.

[3] In the rewriting logic, the equivalent classes can be regarded as states: the term rewriting occurs between the equivalent classes.

An *actor computation* is a transition of actor states by applying the deduction rules based on the following rewrite rules[4].

$$
\begin{aligned}
&\lambda_1 : \langle\, (\texttt{if}\ e_{true}\ e_{false}\ r\ k)\ ,\ \texttt{T}\,\rangle \rightarrow \langle\, k\ ,\ eval(\ e_{true}\ ,\ r\ )\,\rangle \\
&\lambda_2 : \langle\, (\texttt{if}\ e_{true}\ e_{false}\ r\ k)\ ,\ \texttt{F}\,\rangle \rightarrow \langle\, k\ ,\ eval(\ e_{false}\ ,\ r\ )\,\rangle \\
&\lambda_3 : \langle\, (\texttt{app}\ (v\ vs)\ (vs')\ r\ k)\,\rangle \rightarrow \langle\, (\texttt{app}\ (vs)\ (vs')\ r\ k)\ ,\ eval(\ v\ ,\ r\ )\,\rangle \\
&\lambda_4 : \langle\, (\texttt{app}\ ()\ ((\texttt{closure}\ (ss_{args})\ r\ e_{body})\ vs)\ r'\ k)\,\rangle \\
&\qquad\qquad\qquad\qquad \rightarrow \langle\, k\ ,\ eval(\ e_{body}\ ,\ [ss_{args}, vs]/r\ )\,\rangle \\
&\lambda_5 : \langle\, (\texttt{app}\ ()\ (\texttt{list}\ vs)\ r\ k)\,\rangle \rightarrow \langle\, k\ ,\ (vs)\,\rangle \\
&\lambda_6 : \langle\, (\texttt{app}\ ()\ (\texttt{car}\ (v\ vs))\ r\ k)\,\rangle \rightarrow \langle\, k\ ,\ v\,\rangle \\
&\lambda_7 : \langle\, (\texttt{app}\ ()\ (\texttt{cdr}\ (v\ vs))\ r\ k)\,\rangle \rightarrow \langle\, k\ ,\ (vs)\,\rangle \\
&\lambda_8 : \langle\, (\texttt{app}\ ()\ (k\ v)\ r\ k')\,\rangle \rightarrow \langle\, k\ ,\ v\,\rangle \\
&\lambda_9 : \langle\, (\texttt{app}\ ()\ (\texttt{callcc}\ (\texttt{closure}\ (s_{arg})\ r'\ e_{body}))\ r\ k)\,\rangle \\
&\qquad\qquad\qquad\qquad \rightarrow \langle\, (\texttt{halt})\ ,\ eval(\ e_{body}\ ,\ [s_{arg}, k]/r'\ )\,\rangle
\end{aligned}
$$

λ_1 and λ_2 explain the transitions of the conditional expressions. If `T` is passed to the *if* continuation, e_{true} is evaluated; otherwise e_{false} is evaluated. We explain the *eval* operator in the next paragraph. λ_3 shows how the parameters to a function are evaluated sequentially: v, the first yet to be evaluated element, is removed from the continuation and its evaluation is passed to the resulting continuation. When all parameters are evaluated, they are applied to the function. λ_4 to λ_9 explain the parameter applications on different types of functions. λ_4 is for a user defined function. A user defined function is evaluated to be a *closure* structure. Thus, the application of parameters extends the environment with the parameters and evaluates the function body in the extended environment. λ_5, λ_6, and λ_7 are for primary operators. For simplicity, we show only the three primary operators for a list manipulation, but the rests are similar. λ_8 is for a continuation: if the function is a continuation, the parameter is passed to the continuation. Observe that the old continuation k' is ignored. λ_9 explains the `callcc` operator. The parameter to the `callcc` operator is a single parameter function which is evaluated to be a closure structure. In λ_9 the body of the single parameter function is evaluated in the environment extended with the continuation k.

In the rewrite rules above, we used an operator $eval(\ \_\ ,\ \_\ ) : \mathcal{E} \times \mathcal{R} \rightarrow \mathcal{V}$. *eval* evaluates the expression $\mathcal{E}$ in the environment $\mathcal{R}$. The following equations on the terms of the *eval* operator build an equivalent class of terms for the evaluation[5].

$$eval(\ s\ ,\ r\ ) = r[s] \tag{1}$$

$$eval(\ n\ ,\ r\ ) = n \tag{2}$$

[4] We simplified the rules by writing $\langle\, (\texttt{app}\ (vs)\ (vs')\ r\ k)\ ,\ v\,\rangle$ as $\langle\, (\texttt{app}\ (vs)\ (vs'\ v)\ r\ k)\,\rangle$.

[5] The equational rewriting based on these equations on *eval* terms will produce their normal forms.

$$eval(\ k\ ,\ r\ ) = k \tag{3}$$

$$eval(\ (\texttt{closure}\ (ss_{args})\ r\ e_{body})\ ,\ r'\ ) = (\texttt{closure}\ (ss_{args})\ r\ e_{body}) \tag{4}$$

$$eval(\ (\texttt{lambda}\ (\ ss_{args}\ )\ e_{body})\ ,\ r\ ) = (\texttt{closure}\ (ss_{args})\ r\ e_{body}) \tag{5}$$

$$eval(\ (\texttt{quote}\ vs)\ ,\ r\ ) = (vs) \tag{6}$$

$$\langle\ k\ ,\ eval(\ (e_{func}\ es_{param})\ ,\ r\ )\ \rangle = \langle\ (\texttt{app}\ (es_{param})\ ()\ r\ k)\ ,\ eval(\ e_{func}\ ,\ r\ )\ \rangle \tag{7}$$

$$\langle\ k\ ,\ eval(\ (\texttt{cond}\ e_{test}\ e_{true}\ e_{false})\ ,\ r\ )\ \rangle = \langle\ (\texttt{if}\ e_{true}\ e_{false}\ r\ k)\ ,\ eval(\ e_{test}\ ,\ r\ )\ \rangle \tag{8}$$

Equation (1) shows that the evaluation of a symbol is the value looked up from the environment. Specifically, the equation means that the *eval* term and the terms involved in the look up operation are in the same equivalent class. Equation (2) to Equation (4) show that the evaluations of numbers, continuations, and closures are themselves. Equation (5) shows how user defined functions are converted to the closure structures. A closure is a list of the function parameter names, an environment, and the function body. When writing a program, referencing a function itself from its body are often necessary; for example, to make a recursive call. Although one can use the *Y combinator* [43] on the λ expression for this purpose, `rec` operator provides an easy access to the name of a function from its body. The following equation shows what `rec` means in terms of the Y combinator.

$$\begin{aligned}&(\texttt{rec}\ (\ s_{fn}\ ss_{args}\ )\ e_{body}) = (Y\ (\texttt{lambda}\ (s_{fn})\ (\texttt{lambda}\ (ss_{args})\ e_{body}))),\\ &\text{where }\ Y = (\texttt{lambda}\ (\texttt{f})\ (\ (\texttt{lambda}\ (\texttt{y})\ (\texttt{f}\ (\texttt{lambda}\ (ss_{args})\ (\ (\texttt{y}\ \texttt{y})\ ss_{args}))))\\ &\qquad\qquad (\texttt{lambda}\ (\texttt{y})\ (\texttt{f}\ (\texttt{lambda}\ (ss_{args})\ (\ (\texttt{y}\ \texttt{y})\ ss_{args}))))\ ))\end{aligned}$$

However, in the actual implementation, the `rec` term is transformed directly to a `closure` term like Equation (5) and its environment is extended with a mapping from the function name to the closure itself. Equation (6) shows that the evaluation of a `quote`'d list is the list content. The `quote` operator is useful when sending a list to another ActorNet platform; without it, the ActorNet node would regard the list as a function application with the function of the first element and the parameters of the rest of the elements. Equation (7) explains how a function application is converted to the *app* continuation. Similarly, Equation (8) shows how a conditional expression is converted to an *if* continuation. Observe that unlike *app* continuation the two parameters e_{true} and e_{false} of the *if* continuation are not eagerly evaluated: one of them is evaluated based on the evaluation of e_{test}.

Actors coordinate with others through the asynchronous message passing. These interactions are described as transitions of actor configurations which are a "soup" of actor states and messages. Actor configurations make transitions by applying the deduction rules based on the following rewrite rules.

$$\begin{aligned}&\pi_1 : \langle\ k\ ,\ eval(\ (\texttt{par}'\ \langle\ (e\ es)\ ,\ (ns)\ \rangle)\ ,\ r\ )\ \rangle\ \rightarrow\\ &\qquad \langle\ k\ ,\ eval(\ (\texttt{par}'\ \langle\ (es)\ ,\ (ns\ n)\ \rangle)\ ,\ r\ )\ \rangle\ |\ \langle\ (\texttt{halt})\ ,\ eval(\ e\ ,\ r'\ )\ \rangle,\end{aligned}$$

$$\text{where } n \text{ is a fresh actor id and } r' = [\,\texttt{msgq}\;\texttt{id}\,,\;()\;n\,]/r$$

$$\pi_2 : \langle\, (\texttt{app}\;()\;(\texttt{send}\;n\;v)\;r\;k)\,\rangle \;\rightarrow\; \langle\, k\,,\;()\,\rangle \mid (\texttt{mesg}\;n\;v)$$

$$\pi_3 : \langle\, (\texttt{app}\;(vs_{yet})\;(vs_{done})\;r\;k)\,\rangle \mid (\texttt{mesg}\;n\;v) \\ \rightarrow\; \langle\, (\texttt{app}\;(vs_{yet})\;(vs_{done})\;r'\;k)\,\rangle \;\text{ if } r[\texttt{id}] = n,$$

$$\text{where } r = (\texttt{env}\;(ss\;\texttt{msgq}\;\texttt{id}\;ss')\;(vs\;(vs_m)\;n\;vs')), \\ r' = (\texttt{env}\;(ss\;\texttt{msgq}\;\texttt{id}\;ss')\;(vs\;(v\;vs_m)\;n\;vs'))$$

$$\pi_4 : \langle\, (\texttt{if}\;e_{true}\;e_{false}\;r\;k)\,,\;v'\,\rangle \mid (\texttt{mesg}\;n\;v) \\ \rightarrow\; \langle\, (\texttt{if}\;e_{true}\;e_{false}\;r'\;k)\,,\;v'\,\rangle \;\text{ if } r[\texttt{id}] = n,$$

$$\text{where } r = (\texttt{env}\;(ss\;\texttt{msgq}\;\texttt{id}\;ss')\;(vs\;(vs_m)\;n\;vs')), \\ r' = (\texttt{env}\;(ss\;\texttt{msgq}\;\texttt{id}\;ss')\;(vs\;(v\;vs_m)\;n\;vs'))$$

$$\pi_5 : \langle\, (\texttt{halt})\,,\;v\,\rangle \mid c \rightarrow c \;\text{ if } v \text{ is } \mathcal{N}, \mathcal{S}, \mathcal{K}, \mathcal{P}, \text{ or } \mathcal{L} \text{ sort}$$

π_1 shows how an actor creates other actors; `par` operator takes one or more expressions as its parameters and creates new actors for each expression to concurrently evaluate them. The return value from the `par` operator is a list of the new actor ids. To simplify the explanation, we introduced the following two helper equations.

$$eval(\,(\texttt{par}\;es)\,,\;r\,) \;=\; eval(\,(\texttt{par}'\;\langle\,(es)\,,\;()\,\rangle)\,,\;r\,)$$
$$eval(\,(\texttt{par}'\;\langle\,()\,,\;(ns)\,\rangle)\,,\;r\,) \;=\; (ns).$$

π_2 shows that `send` adds a message to the actor configuration. π_3 and π_4 specify that the message in the configuration is added to the message queue of the recipient actor. Finally, π_5 describes the demise of an actor: when an actor computation is completed, its state is removed from the configuration.

6 ActorNet Implementation

Based on the design proposed in Section 3 and the language definition of the previous section, we discuss the issues in implementing the ActorNet runtime platform.

6.1 ActorNet Network Implementation

ActorNet provides a single virtual WSN to actors by connecting physically separated multiple WSNs through the Internet. Recall that the uniform network structure of the virtual WSN is ensured by making the Internet a single hop broadcast network. ActorNet implements two services called *repeater* and *forwarder* to build the uniform network. The repeater bridges the communications between the Ad-Hoc wireless network and the Internet by passing all messages received from one network to the other. Meanwhile, the forwarder provides a single-hop broadcast overlay over the Internet by replicating the messages from each repeater to each of the others connected to it. The net effect of the repeater/forwarder architecture is transforming the individual physically-separated WSNs into a single-hop neighborhood.

Figure 1 shows the repeater/forwarder network architecture of ActorNet. Each repeater has a node called GenericBase through which the repeater can hear from and talk to its WSN. A repeater injects any message it hears from the Internet into its WSN and it sends any message it overhears from its GenericBase to the forwarder. On the other hand, a forwarder is listening to a TCP port for any connections. Once a connection is made, the client is registered to the forwarder until the connection is terminated. In summary, any message overheard by a repeater from its WSN is transmitted to the forwarder and then retransmitted to the other repeaters and ActorNet platforms running on PCs. Finally, the messages sent to the repeaters are injected into their WSNs.

The network bandwidth difference problem is currently handled by placing a large message buffer at the repeaters. The fast messages from the Internet are gathered at the buffer and then slowly retransmitted to the WSNs. However, as the number of clients to the forwarder is increased, the repeaters will constantly send messages to their WSNs. This will increase the chance of a network collision and drain the energy from the nodes near the GenericBase. In addition, the buffering solution is only valid while the input data rate to the buffer is smaller than its output rate. To address these problems, a smarter scheme that makes the repeaters selectively filter the messages can be used. The filtering is based on the actor computation model: when an actor is created, its unique id is known only to its creator, and as the parent or the children send messages with the new actor ids, others can communicate with the newly created actor. Thus, unless a messages with the actor id have passed through a *repeater*, no actors at the other side of the repeater know the existence of the new actor. Because every data is associated with its type in actor language, the actor id checking at the repeater can be effectively done by adding a new type for the actor ids. Observe that the actor ids stored in repeaters can be regarded as the receptionist names and the external actor names of the *actor configuration* [5]. Our actor configuration of Section 5.3 can be augmented with these actor names after this communication optimization is introduced.

6.2 ActorNet Language Implementation

Recall from section 3 that an actor state is a pair of a continuation and a value, and the computation of an actor is a series of actor state transitions made by applying a state's value to its continuation. In ActorNet, these state transitions are implemented by two core methods of actor language interpreter called `eval` and `apply`. `apply` takes the continuation and the value of an actor state as its parameter and produces a new actor state by applying the value to the continuation. `eval` takes an expression and an environment as its parameter and evaluates the expression within the environment. While evaluating an expression, the values of the identifiers are looked up from the environment that actually is a stack of identifier-value pairs. The environment stack is stored at a structure called *closure* when `eval` encounters a function definition, and is extended when the actual function parameters are applied to the function. The stack structure of environment and the use of closure ensure the lexical scoping rule when looking up the identifiers.

The tail recursion is a recursive call that does not necessarily result in a build up of state information on the stack [47]. Because actor programs use recursions for the loops, the tail recursion removal is crucial for actor programs; without it, the stack will grow for any simple loop implementations. ActorNet implemented only the basic tail recursion removal capability: the return addresses of function calls are eliminated, removing unnecessary growth of the stack. Although it is not a fully optimized capability, loops can be effectively replaced with parameterless functions. The return addresses are naturally eliminated through the use of the continuation in the actor computation.

The computation of an actor is explicitly managed as transitions of actor states. This explicit state management leads to a simple and notationally clean implementation of multi-threading capability. Because all the necessary information required to proceed the computation of an actor is stored in the actor state, the context switching is as simple as taking an actor state from a queue of actor states and then applying its value to its continuation. This mechanism is similar to the trampolining technique [47], except that ActorNet schedules the switching and the states are explicitly managed. Observe that the environments play the role of the stack, but they are essentially linked lists, as oppose to linear arrays, built on the virtual memory. This dynamic structure eliminates the stack management during the context switch.

6.3 ActorNet Platform Implementation

The current implementation of the ActorNet platform is implemented in only 30 kB of code and 2 kB 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.

Figure 4 depicts a layered software architecture of ActorNet platform for a sensor node. A module does not know the modules above it, but it has access to all the modules below it, not just the ones immediately below it. In contrast, actors only use the interpreter module. Thus, the implementation details are hidden from the actor programs.

Virtual Memory. ActorNet provides a *virtual memory* (VM) subsystem which uses 64 kB of the 512 kB serial flash as the virtual memory. This address space is efficiently indexed by a 16-bit integer. The virtual address space is divided into 512 pages of 128 bytes each. In addition, 8 pages of SRAM (1 kB) are used as a cache for the virtual memory. While flash is not commonly used as a virtual memory store due to the limitation on the maximum number of writes it supports, typically about a million writes to each location, the relatively slow operating speed of sensor nodes and small data sizes of mobile actors mean that even long-term deployments of wireless sensors are very unlikely to approach this limit.

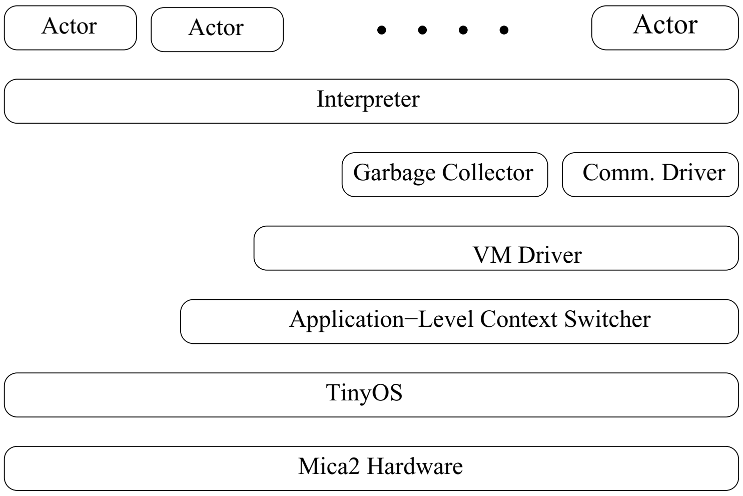


Fig. 4. Software architecture of ActorNet platform (Mica2 node). By making actors interact only with the interpreter layer, the platform differences are hidden from the actors.

An inverted page table 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, the page replacement follows the *Least Recently Used* (LRU) policy. Figure 5 shows the structure of a page. The 128 byte page is divided into a 112-byte data area, a 14-byte bitmap, a 1-bit dirty bit flag, a 4-bit lock count, and 11 bits of reserved space. Because the flash memory writes are slow, we used the dirty bit to avoid an unnecessary page writing. The lock count is used to prevent the VM subsystem from swapping out certain pages. For example, the communication driver of Figure 4 uses a set of static variables defined in a structure called `ComData`. Because this data has buffers shared with the TinyOS communication subsystem, its container page must be locked during transmit and receive operations. This is accomplished by calling the VM's `lock` procedure, subsequently followed by an `unlock` call.

Since there are 112 bytes of data area per page, the effective virtual memory space is 56 kB (512×112). In Figure 5, 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 a disadvantage when searching for a free space, because the VM driver has to load each page from the flash to check the free space. On the other hand, it is crucial to save the precious SRAM space.

Evaluation based on the benchmark of recursively computing the n^{th} Fibonacci numbers showed a page hit ratio of 95.00%. However, the page hit ratio rises to 99.06% if we consider only the page misses involving the flash-write operations.

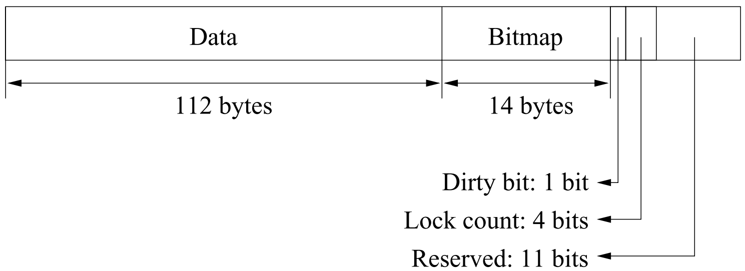
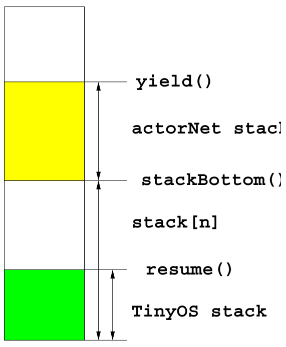


Fig. 5. ActorNet page structure. Including bitmaps in the page structure imposes a performance penalty when searching for a free space. However, having a smaller RAM space than the size of the bitmap, it is an inevitable decision.

Application-Level Context Switching. Figure 6 shows pseudo code of `yield` and `resume` methods for the context switching mechanism. In order to perform the context switching correctly, stack contents and register values must be preserved. We reserved a stack space for TinyOS and other 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 that `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 at the `toApp` variable and the control flow is returned 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.

The left side of Figure 6 shows the stack configuration with this mechanism. In the figure, the stack fills up from the bottom. The shaded area below the `resume()` is the stack space used by TinyOS. The white area below the `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 a stack usage, the applications running on a Mica2 are 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 the `stackBottom()` is the stack space used by the ActorNet platform. The `yield()` line shows the top of the application stack when the `yield` is called.



```
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);
}
```

Fig. 6. Application level context switching mechanism: stack[n] local variable provides a gap between the beginning of ActorNet platform stack (StackBottom) and the stack space for TinyOS. The yield and resume calls switch the stack pointer between these two regions accordingly.

In order to explore the utility of the context switching mechanism, let us consider the following NesC program for the `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, the computation continues from the `yield()` call of the `read` function as if it had just returned from the `yield`. Note that we do not need to divide the application program into two phases as in Figure 2. Hence the yield-resume mechanism improves the maintainability of applications.

Multi-Phase Garbage Collector. We implemented a scalable *mark and sweep* garbage collector [7, 16] to reduce programming errors and to relieve the developers of the burden of manual memory management. Our actor language is a typed language: every data value is tagged with a byte for its type. Because there are only a handful of data types in the actor language, the rest of the bits can be used as marking flags. In fact, the garbage collector uses two bits for its marking and the communication stack machine uses another bit for serialization. Marking the reachability of a memory cell from any active actor states can be done easily because all actor states are explicitly managed. However, there are also temporary data produced by the actor language interpreter that are not yet bound to their state. To prevent them from being swept away, ActorNet manages a list of the temporary data until their actor state is updated.

In our experiments, the conventional mark and sweep GC can take as long as 10 seconds on Mica2 nodes. This delay can slow down the communication speed considerably, as flash write operations prevent any other computations, including radio communication, in TinyOS. Due to the memory limit, we cannot allocate enough communication buffers to cover the full 10 seconds of GC. We could squeeze the memory to make a communication buffer for 4 packets, but with the conventional GC algorithm this buffer can allow only 1 packet per 2.5 seconds. Instead, we redesign the GC algorithm to have a shorter latency.

To solve this problem, we divide the sweep step into several subphases. Each subphase clears 10 pages, which takes approximately 150 ms. If we disregard the mark phase, ideally, we can send as many as 26 packets per second, because ActorNet has a communication buffer for 4 packets. With the multi-phase GC algorithm, there is a transient time that the mark phase is finished, but the sweep phase is not completed for all pages. The memory allocated during the transient time needs a special care. Suppose that we do not mark the freshly allocated memory, then the memories allocated at not-yet-swept pages will be erroneously deallocated later. On the other hand, if we mark the fresh memory, the memories allocated at the already swept page will not be cleared in the next round of GC. To solve this problem, we implement a 2 bit marking scheme. In this scheme, we alternate the marking bit on each GC round and mark all freshly allocated memories with the current mark bit. Then, the freshly allocated memories will not be swept as they are marked, and the marking in the next round can be done correctly as it uses a different marking bit.

Communication Stack Machine. Sending and receiving a structured data involve data serialization and deserialization. Considering the sender/receiver imbalance, we built a stack machine for the communication. A sender traverses a structured data and sends a serialized stream of stack manipulation commands and data. The receivers can then reconstruct the data structure by simply following the stack commands.

A sender uses `encode` method to transmit a serialized stream of stack commands and data, and the receivers use `decode` method to restore the data. Both methods use a stack and an array called `adrsTable` to manage multiply referenced addresses.

The `encode` algorithm is done in two steps. In the first step, `encode` fills the `adrsTable` with the addresses of multiply referenced data. After this step,

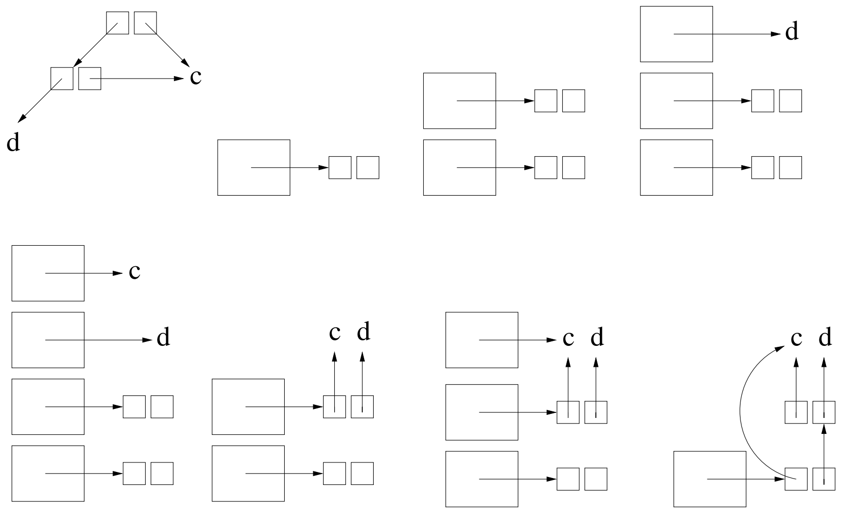


Fig. 7. A data structure to send (the first graph), and the stack configurations of a receiver (the next 7 graphs)

all data reachable from the parameter are marked. In the second step, `encode` generates a stream of stack commands and data while clearing the mark. The details of the second step are as follows:

1. If `encode` visits a marked non-pair data, it sends the type and the value of the data, and clears the mark. When `decode` receives the type tag and the value, it pushes the address of the value.
2. If `encode` visits a marked pair, it sends a `TagPair` tag, processes the two elements of the pair, and sends a `CmdCons` tag. `decode` creates a pair on receiving the `TagPair` as a place holder, and pushes the pair's address. It pops two elements from the stack and links them to the place holder beneath later when it receives the `CmdCons`.
3. If the marked data of cases 1) and 2) are in the `adrsTable`, `encode` sends a `CmdSaveRef` tag and the index of the data in the `adrsTable`. On receiving the `CmdSaveRef`, `decode` stores its top element at the index of its `adrsTable`.
4. Finally, if `encode` visits an unmarked data, it sends a `TagRef` tag and the index of the data in its `adrsTable`. On receiving the `TagRef`, `decode` pushes the address at the index of its `adrsTable`.

As an example of the serialization, suppose that we are sending the first graph of Figure 7. The sequence of data sent from `encode` is `TagPair, TagPair, TagWord, d, TagWord, c, CmdSaveRef, 0, CmdCons, TagRef, 0, CmdCons, CmdEnd`. From this stream of stack commands and data, `decode` replicates the same structure on its side. The 7 graphs from the second graph of Figure 7 show how the receiver stack changes.

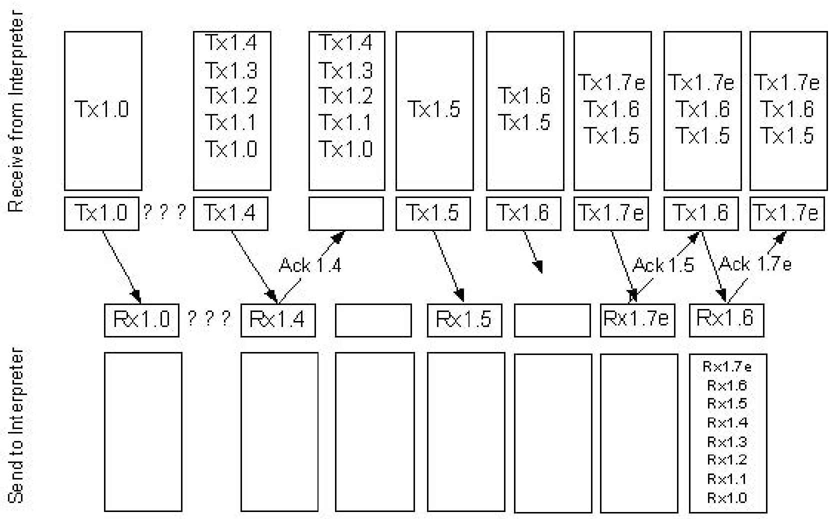


Fig. 8. Selective-repeat protocol for reliable actor migration

Reliable Communication. Execution and communication in a WSN is not always reliable. Many multi-agent applications do not depend on the reliability of a particular actor: the system simply waits for a timeout before launching another instance of a failed actor. Because reliable migration can increase communication cost and latency, the first release of ActorNet did not implement reliable migration. For other applications, however, time-outs to deal with a lack of reliable actor migration simply introduces unacceptable delays.

Two reliable communication methods have been implemented to provide reliable communication. The first is a *selective repeat protocol* based on the *sliding window* concept. This is similar to TCP, where the sender transmits all messages in an actor and then waits for the acknowledgments. Messages that have not been acknowledged are retransmitted. The receiver waits for an entire actor to be transmitted before it is added to the buffer for evaluation and execution. The other method is a simpler *stop-and-wait protocol* which sends one packet at a time and the sender blocks until the receiver acknowledges the packet. Under this protocol, every packet is retransmitted periodically until acknowledged. Figures 8 and 9 illustrate the behavior of these protocols.

We found the stop-and-wait protocol to be superior for environments with low packet loss rates (under 15%). This is because stop-and-wait has lower processor overhead, while selective repeat outperformed significantly in lossier environments. Both implementations are available in the ActorNet platform, and can be selected as appropriate for the application environment.

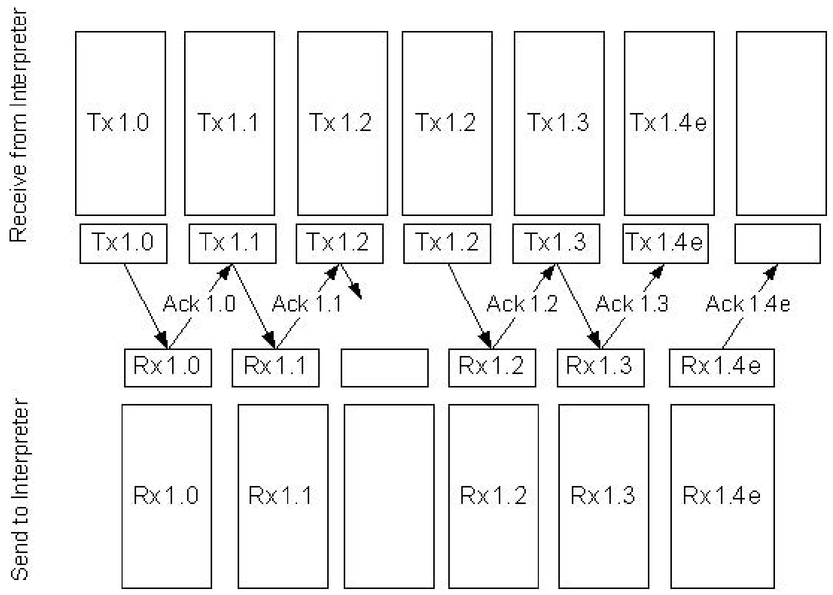


Fig. 9. Stop-and-wait protocol for reliable actor migration

7 Performance Evaluation

We now evaluate the experimental performance of the ActorNet platform. The focus of the experiments described here is to show that the overhead incurred by its constituent services is not prohibitive. Thus, ActorNet is a suitable platform for mobile agents in resource-constrained sensor networks. Our evaluation has three parts:

1. The page hit ratio of the virtual memory subsystem and its impact on system performance.
2. The performance of the multi-phase garbage collector.
3. The communication costs incurred by ActorNet.

7.1 Virtual Memory Performance

We use the benchmark of computing the n^{th} Fibonacci number to evaluate the performance of the VM subsystem. A recursive version of this program is simple, but its exponential behavior is complex enough to carry out a performance evaluation.

As one might expect, as the page cache size increases, the page hit ratio increases. However, in a resource-limited computing environment such as a sensor node, we cannot increase the cache size indefinitely. We must consider a trade-off between the performance and the number of applications that can be run on the

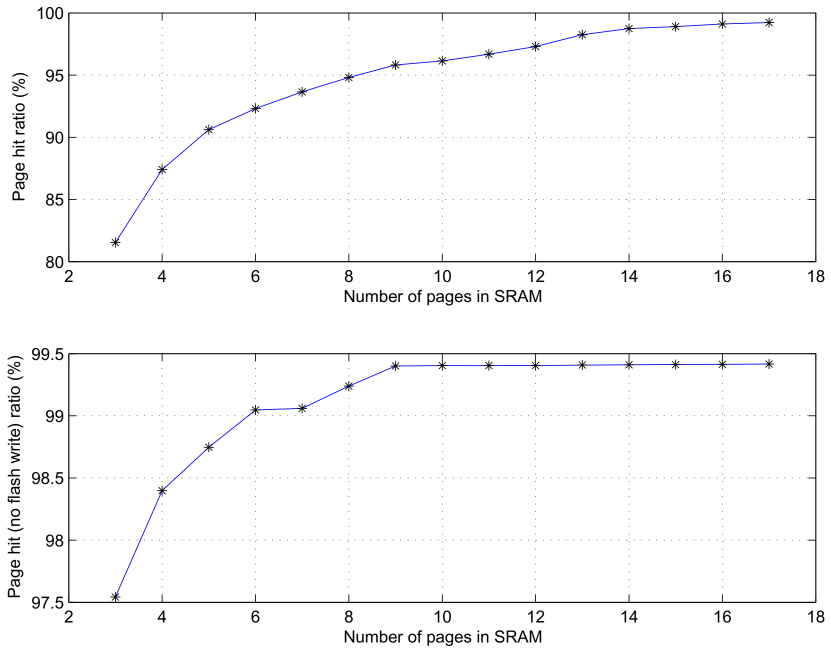


Fig. 10. Page hit ratio (top) and non-flash-write page hit ratio (bottom)

same platform (as not all applications use our VM). The first graph of Figure 10 shows the page hit ratio vs. cache size (number of pages in SRAM). Its shape is approximately concave and increases with cache size. After about 14 pages, the slope is almost flat. However, in the Mica2 platform, the flash write operations dominate the time spent in the VM subsystem. Hence, considering only the flash write operations as page-misses is a more accurate performance measure for the ActorNet platform. The second graph of Figure 10 shows the page hit ratio considering only the flash writes as a 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%.

7.2 Multi-Phase GC Performance

The slow flash write operation of the Mica2 poses a challenge for the garbage collection. As discussed earlier, the GC delay directly limits communication speed. In order to reduce the delay due to GC, we devised a multi-phase GC algorithm. We evaluate the performance of our multi-phase GC as a function of the number of pages swept per phase. The first graph of Figure 11 shows the number of flash

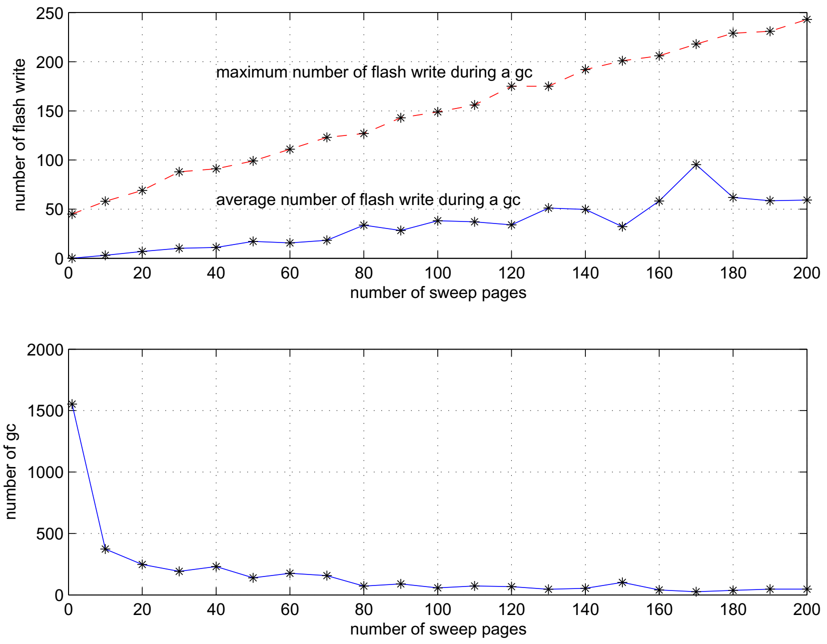


Fig. 11. The number of flash writes during a GC phase (top). The number of GC phases (bottom).

write operations during a GC phase. The solid line shows an average number of flash writes, which can be interpreted as the expected delay due to GC for each phase, and the dashed line shows the maximum number of flash writes, which can be interpreted as the worst case GC delay per phase. The two lines are roughly increasing functions of the number of pages swept, which agrees with intuition. The second graph of Figure 11 shows the number of times GC is called during an experiment. As 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 ms on average, and in the worst case it takes about 870 ms.

There is another merit of the multi-phase GC other than the reduced delay per 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 12 shows the number of mark operations during an experiment. Note that

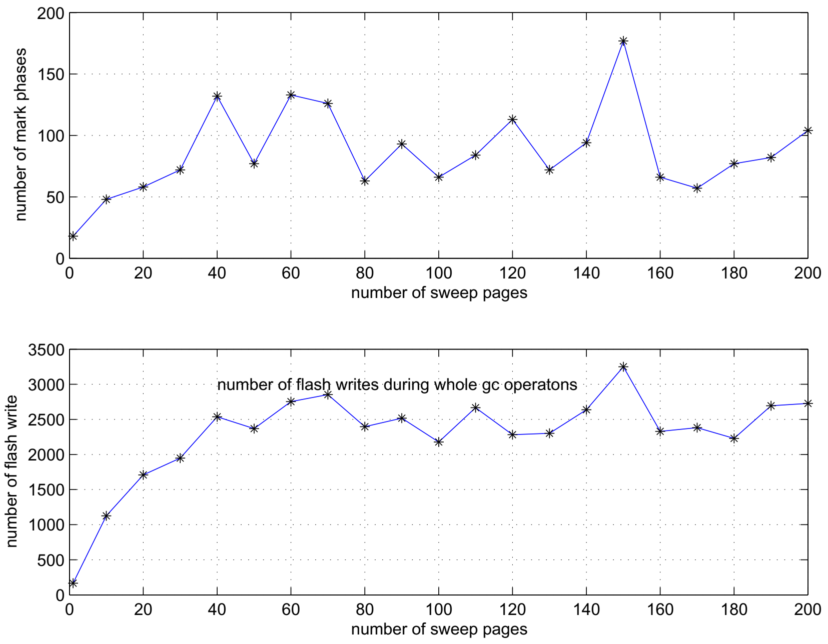


Fig. 12. The number of Mark phases called (top). The number of flash writes due to GC (bottom).

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 12 shows the total number of flash writes made for GC during an experiment. Specifically, it shows an increasing, concave curve: when few pages are swept per phase, the related data tends to aggregate. Thus, related data is more likely to be found in the cache, which reduces the number of flash writes. However, sweeping too few pages at a time results in overly frequent calls to GC, as seen in the second graph of Figure 11.

7.3 Evaluation of Communication Performance

Next we evaluate the communication costs of the example application of Section 4. This application does not require a routing service: 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 the data at a central node. When the gradient path is a straight line, and assuming that the nodes are uniformly distributed, the number of nodes involved in the experiment is proportional to $\sqrt{n}$ for a WSN of n nodes.

Table 1. The number of messages and the size of messages transmitted by the actor program of Figure 3

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

To assess the communication performance, we measured the number and size of messages while running the actor program in Figure 3. Table 1 summarizes the results. Broadcasting a measurement actor to neighboring nodes requires 107 bytes of data in 4 messages. Sending a temperature reading requires 27 bytes, which can be sent in a single message. 27 bytes for a simple temperature reading may look like an overhead. The overhead can be attributed to the type information, the list data structure, and the communication stack commands. However, they are necessary overheads to make actor messages generic and not application dependent. Observe that the measure actor is only 107 bytes long. A similar program that periodically samples the temperature and broadcasts the result is about 28 kB. 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 extra 8 bytes per hop account for the local variables stored during the migration (recursion). Note that as the actor migrates back to the base station, it discards the unnecessary pieces of its code. As such, the returning actor shrinks in size from 1,629+ bytes to 474+ bytes.

8 Case Study: Ambiance Platform

The ActorNet mobile agent framework is used as part of the macroprogramming system called *Ambiance* [42]. The goal of Ambiance is facilitate non-expert programmers in using pervasive computing devices in the environment, including WSNs. In the Ambiance system, users make "ubiquitous queries" called uQuery through a web interface. A uQuery is an aggregation of flow-independent specification of tasks whose comprising steps, such as primitive calls, loops, nested calls, and application-specific constructs, are converted to a task graph of concurrent active objects which can be executed concurrently. This makes Ambiance an *open system* where users and tasks can join or leave the system at any time.

In Ambiance, WSN computations are automatically converted to actor programs and executed on the ActorNet platforms deployed in the sensor network. Ambiance also uses a high level mobile agent system based on the meta-actor architecture by Mechitov et al. [37]. This system is focused on effectively scheduling and sharing middleware services for WSNs. For a given request, it runs matching algorithms on a repository of service implementations and node capabilities to find a feasible set of candidates, and optimally deploy the implementation to a fine-grained set of selected nodes.

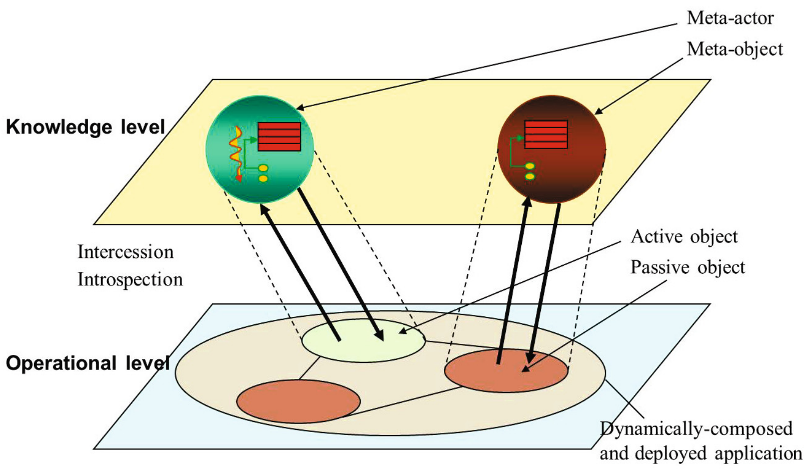


Fig. 13. Two-level adaptive object model architecture for controlling active objects in Ambiance

The Ambiance platform follows the architectural style of *Adaptive Object-Models* (AOM), which define a family of architectures for object-oriented software systems dynamically programmable by domain experts. AOMs are *meta-level architectures* that enforce separation of concerns, and in particular the separation of high-level logic from technical aspects of implementation. In other words, AOMs store the base object as used in the code alongside its meta-data description in terms accessible to the domain expert.

Ambiance further extend this architectural style to enable high-level specifications of global behavior by uncoordinated end-users through a specialized Web interface, and their translation into not only meta-objects, but also meta-*actors*, which control and customize the runtime behavior of both passive and active application objects. These meta-objects are dynamic, they have the capability to observe the application objects and the environment (*introspection*), and to customize their own behavior by analyzing these observations (*intercession*), as seen in Figure 13. This is a form of *reflection*, which allows a program to reason about and affect its own representation and behavior. Watanabe and Yonezawa [54, 57] introduced the notion of reflection in object-oriented concurrent computation model with message passing, which is in many respects similar to the actor model of Ambiance.

The key innovation with respect to the AOM architecture is the separation of the *knowledge level*, where the application, data, service definitions are represented, from the *operational level*, where actual low-level implementation of these objects and services are located and code execution takes place. Figure 14 provides an overview of the system decomposed into these two levels. Note that program representation and transformation environments exist entirely in the

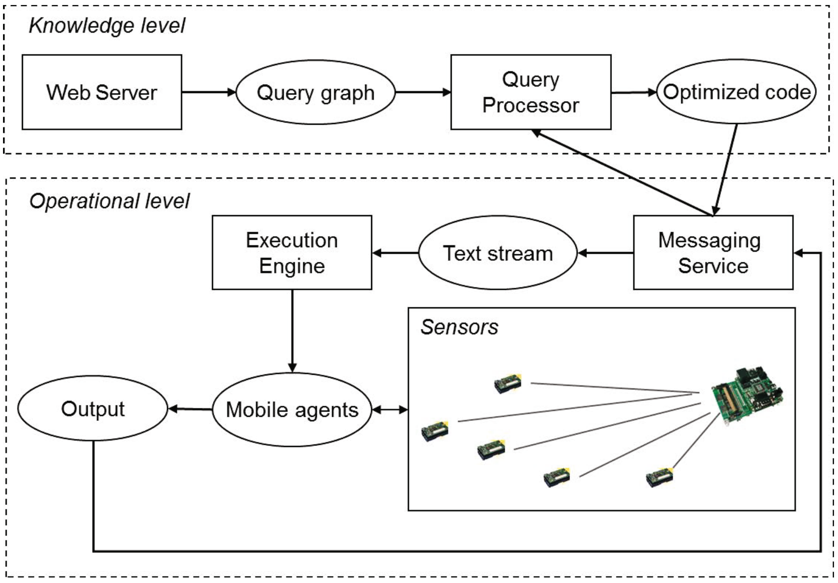


Fig. 14. Ambiance macroprogramming platform runtime

knowledge level, and are thus logically independent of the underlying execution framework used in the deployment environment.

At the operational level, a fine-grained mobile code deployment framework must be available on resource-limited, real-time distributed systems comprising the ambient infrastructure. The mobile code deployment platform is responsible for: 1) deploying and executing dynamically generated low-level code, 2) dynamically discovering and providing access to all sensor and computational resources in the system, and 3) implementing the elements of the service repository. In Ambiance, this role is filled by the ActorNet runtime.

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 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 used in other WSN-based mobile agent implementations. Mobile agents, called base-level actors in Ambiance, are automatically generated using templates in the knowledge level. The entire base-level application is then deployed as a system of cooperating mobile agents in the WSN, where each node is an ActorNet platform.

9 Related Work

There has been related work on WSNs in a number of areas, including mobile agents, intelligent agent systems, and database systems. We discuss this and other systems related work below.

9.1 Mobile Agent Systems for WSNs

Several attempts have been made to implement efficient mobile agent platforms on WSNs. With the proactive mobile agents, the flexibility in reprogramming and operating WSNs, and the energy saving due to the reduced amount of communication can be maximized. Mate [31] is one of the first mobile agent platform designed for WSNs. Sharing some of the same design goals as ActorNet, it is specifically targeted for highly memory restricted sensor nodes: its stack-based virtual machine operates on a Rene2 mote with only 16 kB of program memory and 1 kB of RAM. Mate features high-level instructions that result in a small code size and efficient code migration. Agilla is another mobile agent platform for WSNs [17]. Like Mate, Agilla is a stack-based virtual machine with special instructions for code mobility. Additionally, Agilla supports multiple applications running on a single node and features a Linda-like tuplespace that decouples data from the spatial constraints [12]. Unlike ActorNet, whose agents are written in a high-level language, programmability and code maintainability in these two systems pose a much greater challenge due to the low level of language abstraction.

Considering the programmability, there is a mobile agent platform for WSNs called SensorWare [10] that provides a high-level language abstraction. SensorWare supports an event-based Tcl-like script language. This high level language not only increases the programmability but also reduces the code size: the specific low-level details are removed by the high level of language abstractions. Currently, SensorWare is implemented only on more powerful platforms such as mobile phones or PDAs. However, with its code size of < 180 kB, it may not be directly applicable to current-generation sensor nodes, such as Mica2 or Telos, which have much tighter memory constraints. In contrast, ActorNet implements an interpreter for a high-level language in under 30 kB of code.

9.2 Intelligent Agent Systems for WSNs

Agent systems, in general, are concerned with high level issues such as negotiation or scheduling. Bryan et al. [30] have designed an agent system for target tracking in WSNs that addresses these high-level aspects of the system. In their system, a WSN is divided into non-overlapping regions called sectors, which are managed by statically assigned sector managers. The sector managers dynamically assign track managers which initiate a new target tracking task as new targets are detected. The tasks are described by alternatively selectable sequences of sub-tasks such that a schedulable plan for a new task can be dynamically built from the space of alternative choices of sub-tasks by negotiating the available resources with other task managers. However, specialized to a target tracking application, their system does not offer the flexibility usually associated with mobile agents. For example, a user has to reprogram the entire WSN loaded with the target tracking application to run different applications. Avoiding platform- and application-specific restrictions on the power of mobile agents is one of the distinguishing features of ActorNet, with its actors being able to take advantage of powerful programming abstractions such as higher-level functions and recursion to implement complex behaviors.

9.3 WSNs as a Data Provider

One of the main usages of a WSN is monitoring the area it is deployed. This task can be done by making the sensors push events to the servers or by making servers pull the data from sensors periodically or in response to the user's requests. In the pull model, WSNs can be seen as a data repository. Naturally, DataBase-like approaches have been developed. For example, with TinyDB [32], a user can easily read the sensor data by making a simple SQL-like query. In TinyDB, considering the efficiency, the sensor data are aggregated together on their way back to the base station. However, despite their efficiency and simplicity, the DataBase like approaches usually provide much less flexibility than mobile agent-based approaches such as ActorNet.

The approach TinyDB has taken on WSNs can be seen as a client/server system where the sensor nodes are the servers providing information in response to the requests from a central client. One of the problems, identified by James *et al* [45], in this client/server approach, especially when the server resources are limited, is that the server cannot provide enough interfaces that could satisfy all the requests of the client. Usually the set of the services a sensor node provides are statically determined when the node is deployed, but the kinds of requests to a WSN can dynamically change over time. Hence, the statically determined services may eventually fail to satisfy the dynamically changing requests. Observe that with a small storage and thus having only handful of fixed services, the utility of a WSN becomes worse as the size of a WSN becomes large. In other words, the efficiency and the scalability of a WSN can be restricted with this static approach. A technique called *Remote Evaluation* has been suggested to address this problem [45]. In this technique, a program is sent to a server to be evaluated remotely and the result is sent back to the client. This Remote Evaluation approach not only increases the flexibility of the server but also reduces the amount of the network communication between the server and the client.

Another approach similar to the Remote Evaluation technique is the work of Jagannathan [23]. His work is focused on the definition of languages for coordination in distributed environments. In his work, a continuation is transferred to a remote node instead of a program and its parameters. A continuation sent to a node can locally process the remotely located data to resolve the synchronization issues. Although we do not send continuations for this purpose, our notion of actor migration bears similarity with this mechanism.

9.4 Related Work in Other Aspects

The actors running on ActorNet are implementation of the Actor model. An actor is a self-contained computing element that communicates with other actors by asynchronous messages. The concept of the actors was proposed by Hewitt [21], and formalized as a transition system by Agha [5]. There are many implementations of the Actor systems including the work of Agha et al. [4], where the location of an actor computation is added to the actor programs to enhance the concurrency.

In developing applications for WSNs, reprogramming of sensor nodes has been a big problem that requires considerable amount of time and effort. For this specific problem, a network reprogramming protocol, called Deluge [22], has been developed. The protocol works like a distributed flooding algorithm [49]: each node compares the versions of the advertised images with its own. When a higher version exists it requests and installs the whole image from the winning advertiser. A practical difficulty is that the application images are often larger than the physical memory size. With this protocol, over-the-air reprogramming of a network becomes easy. However, when an upgrade is required on only a few nodes, Deluge is an overkill since it upgrades unnecessary parts of the network also. Moreover, running several distinct applications concurrently on a single network requires the creation of a large image containing all applications.

Since ActorNet was originally proposed in [28], it has been used as a base technology for other applications for WSNs. In the Ambiance system [42] and the shared middleware service system of Mechitov et al. [37], ActorNet serves as an end-computing platform. Karmani and Agha [25] developed a debugging tool for WSN applications based on ActorNet.

10 Future Research Directions

Although ActorNet provides many useful features not found in any previous sensor network programming platform, the current implementation still has several limitations. We describe several open problems.

One of the biggest challenges is *fault tolerance*: as message transmission in WSNs is via local broadcast, we cannot use a simple message acknowledgment mechanism for reliability. Several reliable communication mechanisms for ActorNet have been evaluated in [44], and we are currently investigating techniques to implement an efficient negative acknowledgment-based rebroadcast reliability mechanism.

ActorNet is a bare bones actor system and does not provide coordination mechanisms. The virtual memory and multi-tasking environments provided by ActorNet open the possibility for the more advanced *coordination mechanisms.* These could be as simple as *tuple spaces* [12] and *ActorSpaces* [3], or more complex ones such as *synchronizers* [18], which can be built on top of the existing distributed storage services for sensor networks [36].

Security is another concern for the mobile agents. Mobile agent platforms can prevent malicious agents performing an admission control against signed agent programs. However, this security checking is more challenging in ActorNet because the program is mixed with the states and is changing over time. However, actors work on isolated memory with well-defined and if the runtime ensures that the actor semantics is correctly implemented, security can be enhanced. Another possibility is to use memory management and garbage collection of actors to enhance security by limiting the temporal exposure of a node.

There are many resource management related issues that we have not considered in this work. For example, only a limited number of actors can operate with reasonable performance on as embedded systems have limited processing power and memory. This means that *resource arbitration* is necessary. Such resource arbitration must be *self-evolving* and *adaptive* to enable autonomic functioning of a WSN. In a way, this problem is analogous to the resource arbitration problem in clouds or in enterprise storage systems [55].

We have also not considered *energy consumption*. Energy is a critical constraint in WSNs and requires careful management. Some embedded nodes provide frequency scaling to conserve energy and this can interact with other behaviors of a node, further complicating energy management (e.g. see [39]).

Despite its limitations, we believe ActorNet provides powerful, efficient, scalable, and high level services for developing applications for WSNs. However, further research is needed to facilitate the broader use of actors for building WSN applications.

Acknowledgments. The authors gratefully acknowledge the support of this research by the National Science Foundation under grants CMS 06-00433, CNS 10-35773, NSF 10-35562, and NSF CMMI 09-28886; the Army Research Office under contract W911NF-09-1-0273, and Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The second author was also supported by the Vodafone graduate fellowship. Sergei Shevlyagin, a University of Illinois undergraduate student, contributed to the implementation of ActorNet's reliable communication protocol. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Abelson, H., Dybvig, R.K., Haynes, C.T., Rozas, G.J., Adams, I.N.I., Friedman, D.P., Kohlbecker, E., Steele, J. G.L., Bartley, D.H., Halstead, R., Oxley, D., Sussman, G.J., Brooks, G., Hanson, C., Pitman, K.M., Wand, M.: Revised report on the algorithmic language scheme. SIGPLAN Lisp Pointers IV(3), 1–55 (1991), `http://doi.acm.org/10.1145/382130.382133`
2. ActorNet, `http://osl.cs.illinois.edu/software/actor-net/`
3. Agha, G., Callsen, C.J.: Actorspaces: An open distributed programming paradigm. In: Chen, M.C., Halstead, R. (eds.) PPOPP, pp. 23–32. ACM (1993)
4. Agha, G., Houck, C., Panwar, R.: Distributed execution of actor programs. In: Banerjee, U., Nicolau, A., Gelernter, D., Padua, D.A. (eds.) LCPC 1991. LNCS, vol. 589, pp. 1–17. Springer, Heidelberg (1992)

5. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7, 1–72 (1997)
6. Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y.R., Herman, T., Kulkarni, S.S., Arumugam, U., Nesterenko, M., Vora, A., Miyashita, M.: A line in the sand: A wireless sensor network for target detection, classification, and tracking. Computer Networks, 605–634 (2004)
7. Azatchi, H., Levanoni, Y., Paz, H., Petrank, E.: An on-the-fly mark and sweep garbage collector based on sliding views. ACM SIGPLAN Notices 38(11) (2003)
8. Basharat, A., Catbas, N., Shah, M.: 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)
9. Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., Han, R.: Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. In: Mobile Networks and Applications, pp. 563–579. Kluwer Academic Publishers (2005)
10. Boulis, A., Han, C.C., Srivastava, M.B.: Design and implementation of a framework for efficient and programmable sensor networks. In: International Conference on Mobile Systems, Applications, and Services. USENIX Association
11. Brooks, R.R., Ramanathan, P., Sayed, A.M.: Distributed target classification and tracking in sensor networks. In: Proceedings of the IEEE (2003)
12. Carriero, N., Gelernter, D.: Linda in context. Communications of the ACM 32, 444–458 (1989)
13. Crossbow Technology, Inc., `http://www.xbow.com/`
14. Dorigo, M., Caro, G.D., Gambardella, L.: Ant algorithms for discrete optimization. In: Artificial Life, pp. 137–172 (1999)
15. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: IEEE International Conference on Local Computer Networks, pp. 455–462. IEEE Computer Society (2004)
16. Endo, T., Taura, K., Yonezawa, A.: A scalable mark-sweep garbage collector on large-scale shared-memory machines. In: Proceedings of the IEEE/ACM Conference on Supercomputing (1997)
17. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. Technical Report WUCSE-04-59. Washington University, Department of Computer Science and Engineering
18. Frølund, S., Agha, G.: A language framework for multi-object coordination. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 346–360. Springer, Heidelberg (1993)
19. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: Proceedings of Programming Language Design and Implementation (PLDI) (June 2003)
20. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: International Conference On Mobile Systems, Applications and Services, pp. 163–176. ACM (2005)
21. Hewitt, C.E.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence 8, 323–364 (1977)
22. Hui, J.W., Culler, D.: 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, pp. 81–94. ACM Press (2004)

23. Jagannathan, S.: Continuation-based transformations for coordination languages. In: Theoretical Computer Science, vol. 240, pp. 117–146. Elsevier Science Publishers Ltd. (June 2000)
24. Kamin, S.N.: Programming Languages An Interpreter-Based Approach. Addison Wesley (1990)
25. Karmani, R., Agha, G.: Debugging wireless sensor networks using mobile actors. In: Real-Time and Embedded Technology and Applications Symposium, Poster Abstract (2008), `http://hdl.handle.net/2142/4607`
26. Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., Turon, M.: Health monitoring of civil infrastructures using wireless sensor networks. In: Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, pp. 254–263. ACM, New York (2007), `http://doi.acm.org/10.1145/1236360.1236395`
27. Kwon, Y., Mechitov, K., Sundresh, S., Kim, W., Agha, G.: Resilient localization for sensor networks in outdoor environments. In: International Conference on Distributed Computing Systems, pp. 643–652 (2005)
28. Kwon, Y., Sundresh, S., Mechitov, K., Agha, G.: ActorNet: An actor platform for wireless sensor networks. In: International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1927–1300 (2006)
29. Lajara, R., Pelegri-Sebastia, J., Solano, J.J.P.: Power consumption analysis of operating systems for wireless sensor networks. In: Sensors, vol. 10, pp. 5809–5826. IEEE (2010)
30. Lesser, V., Charles, L., Ortiz, J., Tambe, M.: Distributed sensor networks 15 (2007)
31. Levis, P., Culler, D.: Mate: A tiny virtual machine for sensor networks. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA (October 2002)
32. Madden, S.R., Szewczyk, R., Franklin, M.J., Culler, D.: Supporting aggregate queries over ad-hoc wireless sensor networks. In: Workshop on Mobile Computing and Systems Application (2002)
33. Mainwaring, A., Polastre, J., Culler, R.S.D., Anderson, J.: Wireless sensor networks for habitat monitoring. In: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA) (2002)
34. Maroti, M., Kusy, B., Simon, G., Ledeczi, A.: The flooding time synchronization protocol. In: Sensys (2004)
35. Marti-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Meseguer, J. (ed.) Electronic Notes in Theoretical Computer Science, vol. 4. Elsevier Science Publishers (2000)
36. Mazumdar, S.: Fast range queries using Pre-Aggregated In-Network storage. Masters' thesis, University of Illinois at Urbana Champaign (2004)
37. Mechitov, K., Razavi, R., Agha, G.: Architecture design principles to support adaptive service orchestration in wsn applications. ACM SIGBED Review 4, 37–42 (2007)
38. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
39. Moinzadeh, P., Mechitov, K., Shiftehfar, R., Abdelzaher, T.F., Agha, G., Spencer, B.F.: The time-keeping anomaly of energy-saving sensors: Manifestation, solution, and a structural monitoring case study. In: SECON, pp. 380–388. IEEE (2012)
40. Nagayama, T., Spencer, B.F., Agha, G., Mechitov, K.: Model-based data aggregation for structural monitoring employing smart sensors. In: International Conference on Networked Sensing Systems (2006)

41. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys) (November 2004)
42. Razavi, R., Mechitov, K., Agha, G., Perrot, J.F.: Ambiance: A mobile agent platform for end-user programmable ambient systems. In: Advances in Ambient Intelligence, Frontiers in Artificial Intelligence and Applications, vol. 164, pp. 81–106. IOS Press (2007)
43. Reade, C.: Elements of Functional Programming. Addison-Wesley (1989)
44. Shevlyagin, S., Mechitov, K., Agha, G.: Implementation of fault tolerance in actornet. In: UIUC Department of Computer Science Undergraduate Research Symposium (2008)
45. Stamos, J.W., Gifford, D.K.: Remote evaluation. ACM Transactions on Programming Languages and Systems, 537–564 (1990)
46. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison Wesley (1992)
47. Sussman, H.A.G.J., Sussman, J.: Structure and Interpretation of Computer Programs, 2nd edn. The MIT Press (1996)
48. Tanenbaum, A.S.: Computer Networks, 4th edn. Prentice Hall (2003)
49. Tel, G.: Introduction to Distributed Algorithms, 2nd edn. Cambridge University Press (2001)
50. TinyOS, `http://www.tinyos.net`
51. Tomlinson, C., Kim, W., Scheevel, M., Singh, V., Will, B., Agha, G.: Rosette: An object-oriented concurrent systems architecture. SIGPLAN Notices 24(4), 91–93 (1989)
52. Venkatasubramanian, N., Agha, G., Talcott, C.L.: Scalable distributed garbage collection for systems of active objects. In: Bekkers, Y., Cohen, J. (eds.) IWMM-GIAE 1992. LNCS, vol. 637, pp. 134–147. Springer, Heidelberg (1992)
53. Watanabe, T., Yonezawa, A.: Reflection in an object-oriented concurrent language. In: Meyrowitz, N.K. (ed.) OOPSLA, pp. 306–315. ACM (1988)
54. Watanabe, T., Yonezawa, A.: Reflection in an object-oriented concurrent language. In: Yonezawa, A. (ed.) ABCL: An Object-Oriented Concurrent System. MIT Press (1990)
55. Yin, L., Uttamchandani, S., Palmer, J., Katz, R.H., Agha, G.A.: Autoloop: Automated action selection in the "observe-analyze-act" loop for storage systems. In: POLICY, pp. 129–138. IEEE Computer Society (2005)
56. Yonezawa, A., Shibayama, E., Takada, T., Honda, Y.: Modelling and programming in an object-oriented concurrent language ABCL/1. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming. MIT Press (1987)
57. Yonezawa, A., Watanabe, T.: An introduction to object-based, reflective, concurrent computation. In: Agha, G., Wegner, P., Yonezawa, A. (eds.) Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming (1988)