13

Thin Middleware for Ubiquitous Computing

Koushik Sen
Gul Agha

13.1 INTRODUCTION

As Donald Norman put it in his popular book *Invisible Computers* [12], "a good technology is a disappearing technology." A good technology seamlessly permeates into our lives in such a way that we use it without noticing its presence. It is invisible until it is not available. Ever since the invention of microprocessors, many computer researchers have strived to make computer technology a "good" technology.

Advances in chip fabrication technology have reached a point where we can physically make computing devices disappear. Bulkier machines have given way to smaller yet more powerful personal computers. It has become possible to implant a complete package of a microprocessor with wireless communication, storage, and a sensor on a cubic millimeter silicon die [8]. Specialized printers print out computer chips on a piece of plastic paper [6]. Computer chips are woven into on a piece of fabric [9]. "Smart labels" (a.k.a passive RFID tags) [7] will soon be attached to every product in the market.

The growth of devices with embedded computers will provide task-oriented, simple services which are highly optimized for their operating environment. More user oriented, human friendly services may be created by networking the embedded nodes, and coordinating their software services.

Composing existing component services to create higher-level services has been promoted by CORBA, DCOM, Jini, and similar middleware platforms. However, these middleware services were designed without paying much attention to resource management issues pertinent to embedded nodes: the middlewares tend to have large

footprints that do not fit into the typically small memories of tiny embedded computers. Thus there is a need for a middleware which allows components to be glued together without a large overhead.

Embedded nodes are autonomous and self contained. They have their own state and a single thread of control, and a well defined interface for interaction. Interaction between nodes is asynchronous in nature. The operating environment for these devices may be unreliable. For these reasons, the interaction between different devices must be arms-length – the failure of one device should not affect another. For example, consider an intelligent home where the clock is networked to the coffee maker and an alarm also triggers the coffee maker. An incorrectly operating coffee maker should not cause the alarm clock to fail to operate. The autonomy and asynchrony in the model we describe helps ensure such fault containment.

Embedded nodes are typically resource constrained – they have small communication range, far less storage, limited power supply, etc. These resource limitations have a number of consequences. Small memory means that not every piece of code that may be required over the lifetime of a node can be pre-loaded onto the node. Limited power supply means that certain abstractions, such as those requiring busy waiting to implement, may be too expensive to be practical.

We propose *thin middleware* as a model for network-centric, resource-constrained embedded systems. There are two aspects to the thin middleware model. First, we represent component services as *Actors* [1, 3]. The need for autonomy and asynchrony in resource-constrained networks of embedded nodes makes Actors an appropriate model to abstractly represent services provided by such systems. Service interactions in the thin middleware are modeled as asynchronous communications between actors. Second, we introduce the notion of meta-actors for service composition and customization. Meta-actors represent system level behavior and interact with actors using an event-based signal/notify model.

13.2 ACTORS

Actors [1, 3] were developed as a basis for modeling distributed systems. An actor encapsulates a state, a set of procedures which manipulate the state, and a thread of control. Each actor has a unique *mail address* and a *mail buffer* to receive messages. Actors compute by serially processing messages queued in their mail buffers. An actor waits if its mail buffer is empty. Actors interact by sending messages to each other.

In response to a message, an actor carries out a local computation (which may be represented by any computer program) and three basic kinds of actions (see Figure 13.1):

Send messages: an actor may *send* messages to other actors. Communication is point-to-point and is assumed to be weakly fair: executing a *send* eventually causes the message to be buffered in the mail queue of the recipient. Moreover, messages are by default asynchronous and may arrive in an order different from the one in which they were sent.

Create actors: An actor may *create* new actors with specified behaviors. Initially, only the creating actor knows the name of the new actor. However, actor names are first class entities which may be communicated in messages; this means

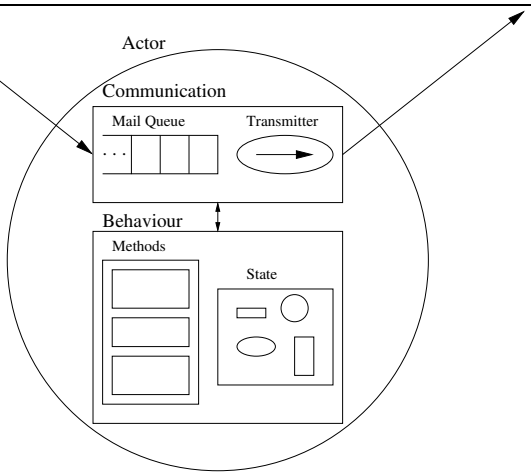

Fig. 13.1 **Actor Model:** Actors are concurrent objects that interact via asynchronous messages.

that coordination patterns between actors may be dynamic and the system is extensible.

Become ready to accept a message: The actor becomes *ready* to process the next message in its mail queue. If there is no message in its mail queue, the actor waits until a new message arrives and processes it.

Asynchronous message passing is the distributed analog of method invocation in sequential object-oriented languages. The *send* and *create* operations can be thought of as explicit requests, while the *ready* operation is implicit at the end of a method. That is, actors do not explicitly indicate that they are ready to receive the next message. Rather, the system automatically invokes *ready* when an actor method completes.

Actor computations are abstractly represented using *actor event diagrams* as illustrated in Figure 13.2. Two kinds of objects are represented in such diagrams: actors and messages. An actor is identified with a vertical line which represents the life-line of the actor. The darker parts on the line represent the processing of a message by the actor. The actor may create new actors (dotted lines) and may send messages (solid lines) to other actors. The messages arrive at their target actors after arbitrary but finite delay and get enqueued at the target actor's mail queue.

Note that the nondeterminism in actor systems results from possible shuffles of the order in which messages are processed. There are two causes of this nondeterminism. First, the time taken by a message to reach the target actor depends on factors such as the route taken by the message, network traffic load, and the fault-tolerance protocols used. Second, the order in which messages are sent may itself be affected by the processing speed at a node and the scheduling of actors on a given node. Nondeterminism in the order of processing messages abstracts over possible communication and scheduling delays.

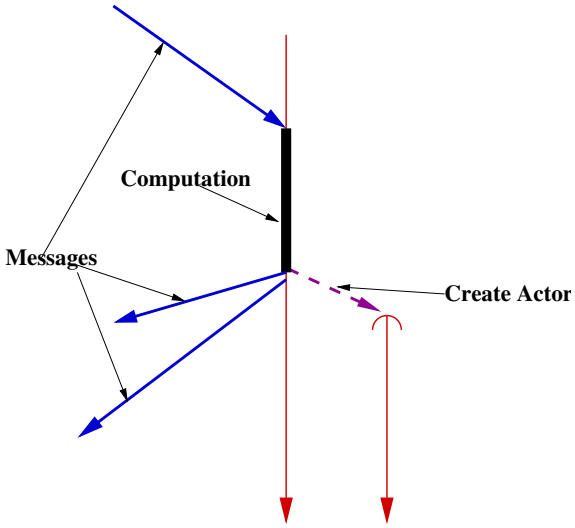

Fig. 13.2 **Actor event diagram**

The nondeterministic model of concurrency provides a loose specification. Properties expressed in this model state what *may*, or what *must*, eventually happen. In reality, the probability that, for example, a message sent at a given time will be received after a million years is practically infinitesimal. One way to express constraints on the arbitrary interleavings resulting from a purely nondeterministic model is by using a probabilistic model. In such a model, we associate a probability with each transition which may depend on the current state of the system. Our specifications then say something about when something may happen with a given probability. We discuss a probabilistic model in some more detail below.

13.2.1 Probabilistic Discrete Real-time Model

Traditional models of concurrent computation do not assume a unique global clock – rather each actor is asynchronous (for example, see [3, 2]). However, when modeling interaction of the physical world with distributed computation, it is essential to consider guarantees in real-time. Such guarantees are expressed in terms of a unique global time or wall clock and the behavior of all devices and nodes is modeled in terms of this reference time (for example, see [11, 14, 13, 10]). This amounts to a synchronous model of actors and it implies a 'tight coupling' in the implementation of actors; network and scheduling delays, as well as clock drift on the nodes, must be severely restricted.

In network embedded systems, a number of factors make a tight coupling in the implementation of actors infeasible. For example, the operation of some embedded devices may be unreliable, and message delivery may have nondeterministic delays due to transmission failures, collisions, and message loss. So in large network embed-

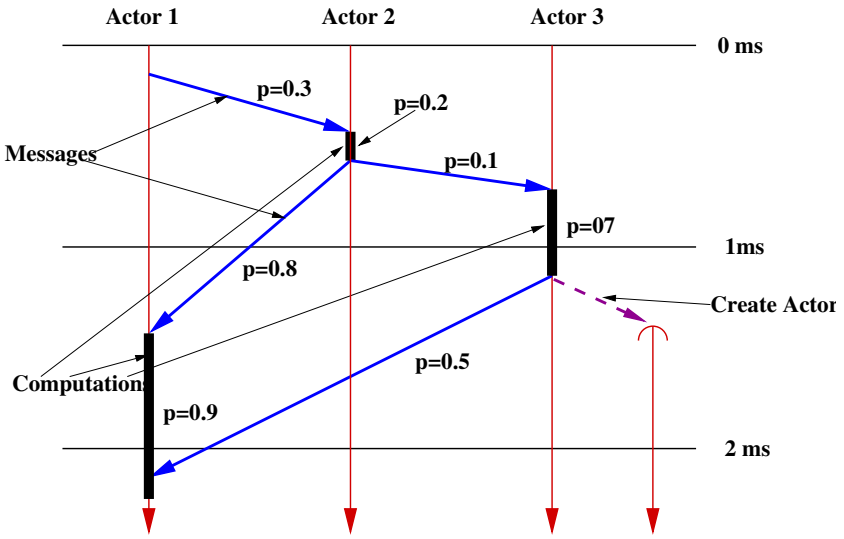

Fig. 13.3 **Probabilities** to capture nondeterminism in computation and communication

ded systems, it is not feasible to maintain a unique reference clock. The introduction of probability in the operations can be thought of as an intermediate synchronization model. In a probabilistic model, we assume that the embedded nodes agree on a global clock, but their drift from the clock is only probabilistically bound. Such probabilities replace the qualitative nondeterminism in computation and communication.

We assume the actors and the messages in transit form a *soup*. The components of the system follow a reference clock with some probability. The global time of the whole system (soup) advances in discrete time steps. The time steps can be compressed and stretched, depending on the kind of property we want to express. For example, the time step can be set to one second or it may be set to one millisecond. The global time of the system advances by one step when all the *actions* (computation and communication) that are possible in that time step have happened (see Figure 13.3). We associate a local clock with each actor and it advances with every global time step. However, it is reset to zero when the actor consumes a message. The clock remains zero when the actor is idle.

At a given time step an actor may be in one of three states:

- ready to process a message from its mail queue,
- busy computing, or
- waiting, because there is no message in mail queue.

If the actor is in either of the first two states, it can take the following actions:

- *complete* the computation in its current time step; or,
- *delay* its computation by one time step.

In the first case, the local clock of the actor remains same and so it is open to other actions in that time step. However, for the second action the local clock of the actor advances by one time step and hence, all possible actions of that actor get disabled for that time step. The two actions get enabled once the global time advances to the next time step. As a function of the state of the system, we associate different probabilities with each of the two actions.

Similarly, at a given time step a message can take three actions:

- it can get *enqueued* at the target actor,
- it can get *lost* and thus removed from the soup, or
- it can get *delayed*, in transit, by one time step.

If a message is delayed in transit, the local time of the message advances by one time step and so the message cannot take any more actions in that global time step. However, all the three actions will get enabled once the global time advances to the next step. Probabilities are associated with each action. The probabilities depend on factors such as the message density in the route taken by the message, the time for which it has been delayed (value of local clock of the message), and the number of messages sharing the same communication channel.

A computation path is defined as a sequence of states that the system has seen in the course of its computation. Note that the system retains the same computation paths as it would have in a nondeterministic model of concurrency. The probability that a particular finite sequence of states in a path will occur is obtained by multiplying the probabilities of all the actions in that sequence of states. Some of these probabilities will grow sufficiently small that they will no longer be relevant to the proof of some properties of our interest.

Using the above model, we can express properties of the form: "Within time t, the system will reach a state which satisfies a property π with probability p." For example:

- the alarm clock will ring at 7:00 a.m. with probability 0.99.
- the microwave will complete popping 95% of the popcorn by 10 a.m. with probability 0.98.

In implementing probabilistic timing specifications, one constrains the system level behavior which involves networks of heterogeneous nodes. A middleware provides a uniform interface to access such nodes. We represent the middleware itself as a collection of actors. The model we describe enables dynamic customizability of the execution environment of an actor in order to satisfy properties such as timing and security.

13.3 REFLECTIVE MIDDLEWARE

A key requirement for middleware is that it must enable dynamic customization – so that services can be pushed in and pulled out at runtime. This scheme of pushing-in and pulling-out of services allows the middleware to keep on a node only those services that are required by an application. The result is a light weight middleware.

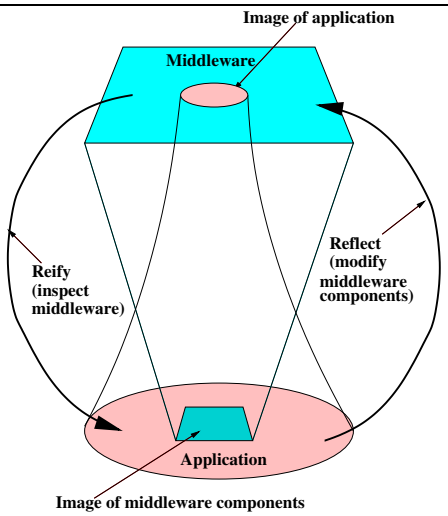

Fig. 13.4 **Reflection:** Application can inspect and modify middleware components.

Because an application may be aware of system level requirements for timing, security, or messaging protocols, it needs to have access to the underlying system. We support the ability of an application to modify its system level requirements by dynamically changing the middleware through the use of *computational reflection*.

A reflective middleware provides a representation of its different components to the applications running on top of it. The applications can *inspect* this representation and modify it. The modifications made to the components are immediately mirrored to the application. In this way, applications can dynamically customize the different components of the middleware through reflection (see Figure 13.4).

We use the *meta-actor* extension of actors to provide a mechanism of architectural customization [5]. A system is composed of two kinds of actors: base actors and meta-actors. Base actors carry out application-level computation, while meta-level actors are part of the runtime system (middleware) that manages system resources and controls the base-actor's runtime semantics.

13.3.1 Meta-architecture

From a systems point of view, actors do not directly interact with each other: instead, actors make *system method* calls which request the middleware to perform a particular action. A system method call which implements an actor operation is always 'blocking': the actor waits till the system signals that the operation is complete. Middleware components which handle system method calls are called *meta-actors*. A meta-actor executes a method invoked by another actor and returns on the completion of the execution. The requisite synchronization between an actor and its meta-actor is facilitated by treating the meta-actor as a passive object: it does not have its own thread of control. Instead, the calling object is suspended. In other words, an actor and its

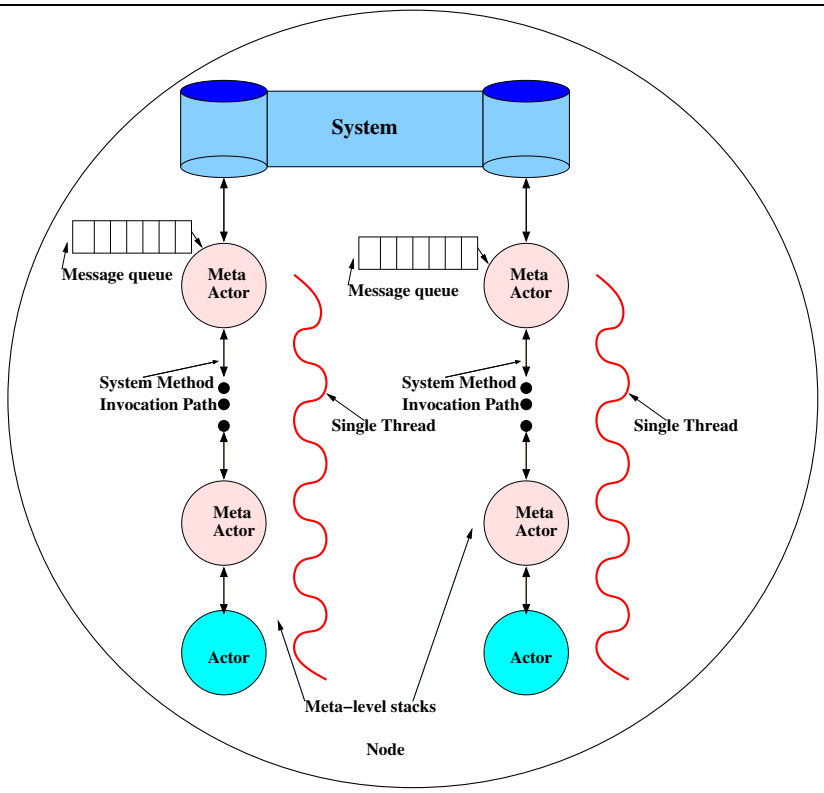

Fig. 13.5 **Stacks of meta-level actors in an embedded node.**

meta-actor are not concurrent – the latter represents the system level interpretation of the behavior of the former.

A meta-actor is capable of customizing the behavior of another actor by executing the method invoked by it. An actor customized in this fashion is referred to as the *base actor* relative to its meta-actor. To provide the most primitive model of customization a meta-actor can customize a single base-actor. However, multiple customizations may be applied to a single actor by building a *meta-level* stack, where a meta-level stack consists of a single actor and a stack of meta-actors (see Figure 13.5). Each meta-actor customizes the actor which is just below it in the stack. Messages received by an actor in a meta-level stack are always delegated to the top of the stack so that the meta-actor always controls the delivery of messages to its base-actor. Similarly messages sent by an actor pass through all the meta-actors in the stack.

We identify each operation of a base-actor as a system method call as follows.

- **send(*msg*):** This operation invokes the system method `transmit` with `msg` (`msg` is the message sent by the actor) as argument. If the actor has a meta-actor on its top it calls the transmit method of the meta-actor and wait for its

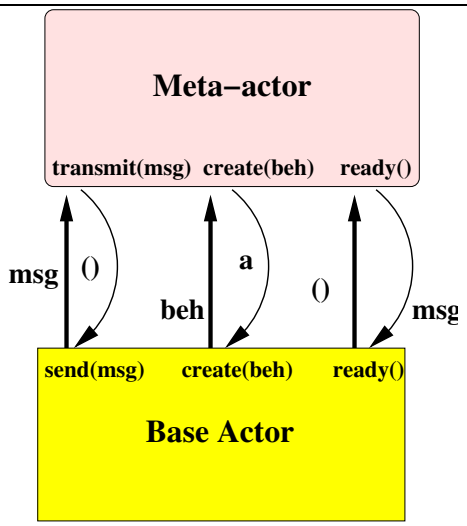

Fig. 13.6 **Interaction between meta-actor and base actor**

return. The method returns without any value. Otherwise, if the actor is not customized by a meta-actor, it passes the message to the system for sending.

- **create(*beh*):** This operation invokes the system method `create` with the given `beh` (`beh` is the behavior with which the newly created actor will be instantiated) as argument. If there is a meta-actor on top of the actor, it calls the create method of the meta-actor and waits for its return. The method returns the address `a` of the new actor. Otherwise, the actor passes the create request to the system.

- **ready():** The system method `ready` is invoked when an actor has completed processing the current message and is waiting for another message. If the actor has a meta-actor on top it calls the ready method of the meta-actor and waits for its return. The method returns a message to the base-actor. Otherwise, the actor picks up a message from its mail queue and processes it. Notice, there is a single mail-queue for a given meta-level stack.

The method call-return mechanism for different actor operations and the availability of a single queue for a meta-level stack makes the execution of a meta-level stack single threaded. So explicit scheduling of each actor in the stack is not required. The meta-actors behave as reactive passive objects which respond only when a system method is invoked by its base actor. The single thread implementation of a meta-level stack is important, as most of the embedded devices can have a single thread only. An example of such a embedded OS is TinyOS which runs on motes.

Every meta-actor has a default implementation of the three system methods. These implementations may be described as follows:

- **transmit(*msg*):** If there is a meta-actor on its top, it calls `transmit(msg)` method of that meta-actor and waits for it to return. Otherwise, it asks the system to send the message to the target and returns.
- **create(*beh*):** If the actor has a meta-actor at its top, it calls `create(beh)` method of that meta-actor and waits for the actor to return with an actor address. Otherwise, the actor passes the create request to the system and waits till it gets an actor address from the system. After receiving new actor address, the actor returns it to the base actor.
- **ready():** If there is a meta-actor on top of it, it calls `ready()` method of that meta-actor and waits for it to return a message. Otherwise, the actor, by definition located at the top of the meta-level stack, dequeues a message from the mail queue. After getting the message, the actor returns the message to the base actor.

```
actor Encrypt(actor receiver) {
    // Encrypt outgoing
    // messages if they
    // are targeted to
    // the receiver
    method transmit(Msg msg) {
        actor target = msg.dest;
        if (target == receiver)
            target ← encrypt(msg);
        else
            target ← msg;
        return;
    }
}
```

```
actor Decrypt() {
    // Decrypt incoming messages
    // targeted for
    // base actor (if necessary)
    method ready() {
        Msg msg = ready();
        if (encrypted(msg))
          return(decrypt(msg));
        else
          return(msg);
    }
}
```

Fig. 13.7 **Meta-Level Implementation of Encryption:** The `Encrypt` meta-actor intercepts `transmit` signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the receiver (via the `rcv` method) and, if necessary, decrypts an incoming message before delivering it.

As an example of how we may customize actors under this model, consider the encryption of messages between a pair of actors. Figure 13.7 gives pseudo-code for a pair of meta-actors which may be installed at each endpoint. The `Encrypt` meta-actor implements the `transmit` method which is called by the base-actor while sending a message. Within `transmit`, a message is encrypted before it is sent to its target. The `Decrypt` meta-actor implements the `ready` method which is called when the base actor is ready to process a message. Method `ready` decrypts the message before returning the message to the base-actor.

The abstraction of the middleware in terms of meta-actors gives the power of dynamic customization. Meta-actors can be installed or pulled out dynamically. This pushing in and pulling out of meta-actors by the application itself makes it capable of customizing the middleware. It also makes it possible to have only those middleware components which are required by services of the current application – facilitating our goal of thin middleware.

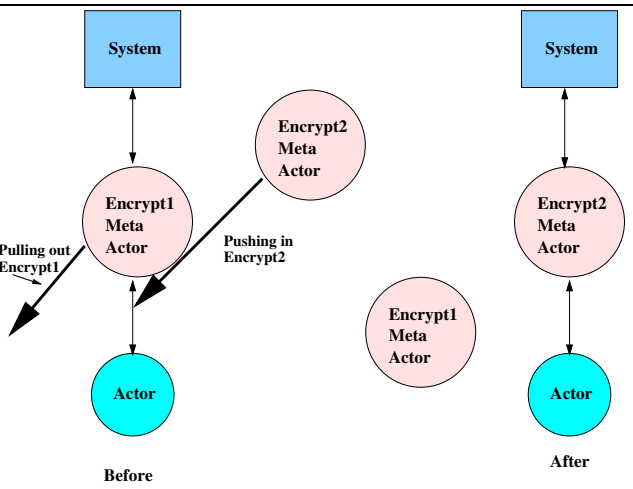

Fig. 13.8 **Dynamic Customization:** Pulling out and pushing in a new meta-actor for the implementation of encryption algorithm

13.4 DISCUSSION

We have described some preliminary work on a model of reflective middleware. We believe that further development of thin middleware will be central to the future integration of computing and the physical world [4]. However, many important problems have to be addressed before such an integration can be realized. We describe two areas to illustrate the problems. These areas relate, respectively, to the model and implementation of middleware.

A formal model of the interaction of the properties of actors and meta-actors has been developed in terms of a *two-level semantics* [15]. This model needs to be extended to its probabilistic real-time counterpart. For example, methods for composition of transition probabilities for actors and their meta-actors have not been developed.

More research is required in the implementation of thin middleware. Current implementation of reflective actor middleware has been based on high-level languages – which necessarily assume a large infrastructure. An alternate implementation would be in terms of a very efficient and small virtual machine which allows enforcement of timing properties. Related problems are incrementally compiling high-level code to such a virtual machine and supporting the mobility of actors executing on the virtual machine.

In our view, the solution to these and related problems define an ambitious research agenda for the coming decade.

13.5 ACKNOWLEDGMENTS

The research described here has been supported in part by the Defense Advanced Research Projects Agency (Contract numbers: F30602-00-2-0586 and F33615-01-C-1907). We would like to thank Nadeem Jamali and Nirman Kumar for reviewing previous versions of this paper and giving feedback.

REFERENCES

1. G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

2. G. Agha. Modeling Concurrent Systems: Actors, Nets, and the Problem of Abstraction and Composition. In *17th International Conference on Application and Theory of Petri Nets*, Osaka, Japan, June 1996.

3. G. Agha, I. A. Mason., S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

4. Gul A. Agha. Adaptive middleware. *Communications of the ACM*, 45(6):30–32, June 2002.

5. M. Astley and G. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Proceedings of the Sixth International Symposium of Foundations of Software Engineering*, pages 1–9, 1998.

6. S. B. Fuller, E. J. Wilhelm, and J. M. Jacobson. Ink-jet printed nanoparticle microelectromechanical systems. *Journal of Microelectromechanical Systems*, 11(1):54–60, 2002. http://www.media.mit.edu/molecular/projects.html.

7. AIM. Inc. http://www.aimglobal.org/technologies/rfid/.

8. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile Networking for Smart Dust. In *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, Seattle, WA, August 1999. http://robotics.eecs.berkeley.edu/ pister/SmartDust/.

9. MIT Media Lab. http://lcs.www.media.mit.edu/projects/wearables/.

10. B. Nielsen and G. Agha. Semantics for an Actor-Based Real-Time Language. In *4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS). Submitted*. Naval Surface Warfare Center Dahlgren Division/IEEE, April 1995. In conjunction with 10th IEEE Int. Parallel Processing Symposium (IPPS), Honolulu, Hawaii, USA.

11. B. Nielsen and G. Agha. Towards reusable real-time objects. *Annals of Software Engineering: Special Volume on Real-Time Software Engineering*, 7:257–282, 1999.

12. D. Norman. *The Invisible Computer*. The MIT Press, Cambridge, MA, USA, 1998.

13. S. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997. PhD. Thesis.

14. S. Ren, G. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36(1):4–12, 1996. Also published in *School on Embedded Systems, European Educational Forum 1996*, pp 52–72.

15. Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *Symposium on Principles of Distributed Computing*, pages 144–152, 1995.

Author(s) affiliation:

- **Koushik Sen, and Gul Agha**

  Department of Computer Science
  University of Illinois at Urbana-Champaign
  Email: [ksen,agha]@uiuc.edu