ProSe: A Programming Tool for Rapid Prototyping of Sensor Networks

Extended Abstract

Mahesh Arumugam
Cisco Systems, Inc.
San Jose, CA 95134
maarumug@cisco.com

Sandeep S. Kulkarni
Michigan State University
East Lansing, MI 48823
sandeep@cse.msu.edu

I. Introduction

Most of the existing platforms (e.g., nesC/TinyOS [1]) for developing sensor network programs use *event-driven programming model* [2]. As identified in [2], [3], while an event-driven platform has the potential to simplify concurrency by reducing race conditions and deadlocks, the programmer is responsible for stack management and flow control.

To simplify programming sensor networks, several *macro-programming* primitives are proposed (e.g., [4], [5], [6], [7], [8], [9]). However, they have the following important drawbacks: (1) the designer of such primitives has to still implement them in a typical sensor network platform (e.g., nesC/TinyOS), (2) if existing primitives are insufficient or need to be extended to deal with hardware/software developments then *domain experts* have to rely on *experts in sensor networks*, and (3) most of these primitives still require the programmer to specify protocols in nesC/TinyOS platform.

In this paper, we focus on application of abstract network protocols/distributed programs towards prototyping sensor networks. Such abstract programs exist for several applications, e.g., leader election, routing, etc. These protocols are often specified in terms of simple event-driven actions where the program responds to actions in the environment or previous actions taken by the program. Due to abstract nature of these programs, they are easy to specify, verify and manipulate.

However, they cannot be applied directly in sensor networks as the model of computation in sensor networks (*write all with collision*) differs from that (*read/write* or *shared-memory*) used in abstract protocols. To deal with this problem, in this paper, we propose *ProSe*, a programming tool for sensor networks. ProSe enables the developers to (1) specify protocols in simple, abstract models (e.g., read/write model, shared-memory model), (2) reuse existing fault-tolerant/self-stabilizing protocols from the literature in the context of sensor networks, and (3) automatically generate and deploy code.

ProSe helps overcome deficiencies of existing event-driven programming platforms for sensor networks (e.g., nesC/TinyOS) that require the programmers to deal with several challenges including buffer management, stack management, and flow control. We expect that ProSe will enable the developers to rapidly prototype and quickly deploy protocols.

II. ProSe: Overview and Case Study

In this section, we present an overview of ProSe and discuss case studies on rapid prototyping of sensor network protocols.

A. Program Structure and Computation Models

The programs in ProSe are specified in terms of guarded commands; each command/action is of the form:

$$guard \longrightarrow statement,$$

where $guard$ is a predicate over program variables, and $statement$ updates program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action, st is executed. A computation model limits the variables that an action can read and write. Towards this end, we split the program actions into a set of processes (i.e., sensors). Each action is associated with one of the processes in the program.

- *Shared-memory model.* In this model, in one atomic step, a sensor can read its state as well as its neighbors and write its own (*public* and *private*) variables.
- *Read/Write model.* In this model, in one atomic step, a sensor can either (1) read the state of one of its neighbors, or (2) write its own variables.
- *Write all with collision (WAC) model.* In this model, each sensor consists of write actions (to be precise, write-all actions). Specifically, in one atomic action, a sensor can update its own state and the state of all its neighbors. However, if two or more sensors simultaneously try to update the state of a sensor, say k, then the state of k remains unchanged. Thus, this model captures the fact that a message sent by a sensor is broadcast. But, if multiple messages are sent to a sensor simultaneously then, due to collision, it receives none.

The WAC model can be effectively used to model computations in sensor networks. Recently, approaches have been proposed for transforming programs into WAC model. They can be classified as: (a) TDMA based deterministic transformation [10] and (b) CSMA based probabilistic transformation [11]. In TDMA based transformation, in WAC model, each sensor executes the actions for which the corresponding guard is enabled in the TDMA slots assigned to that sensor. And, each sensor writes (broadcasts) its state to all its neighbors

in its TDMA slots. In [11], *cached sensor transform* (CST) is proposed that allows one to correctly simulate a program written for shared-memory model in sensor networks.

B. Programming Architecture

The programming architecture of ProSe is shown in Figure 1. ProSe automatically transforms the input guarded commands program into a program in WAC model. Then, ProSe generates the corresponding nesC/TinyOS code. Furthermore, ProSe *wires* the generated code with a MAC layer to implement the write-all action in the WAC model. The MAC layer provides an interface for broadcasting (i.e., writing all neighbors) and receiving WAC messages. Now, the designer can use the TinyOS platform to build the binary of the nesC code. This binary image can then be disseminated across the network using a network reprogramming service. (We refer the reader to [12] for nesC code snapshots generated by ProSe.)

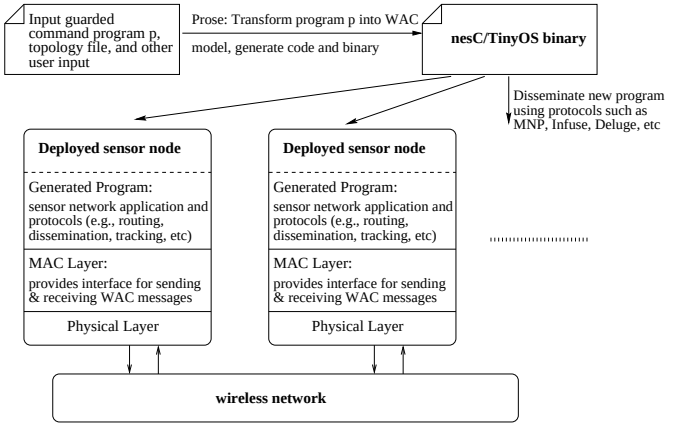

Fig. 1. Programming architecture of ProSe

C. Case Study

In this section, we illustrate ProSe to generate network and application services for sensor networks.

1) Routing Tree Maintenance Program (RTMP): We specify RTMP in shared-memory model in ProSe as shown in Figure 2 [1]. This program is based on logical grid routing program proposed in [14]. In this program, sensors are arranged in a logical grid. The program constructs a spanning tree with the base station as the root. The base station is located at $\langle 0, 0 \rangle$ of the logical grid. Each sensor classifies its neighbors as *high* or *low* neighbors depending on their (logical) distance to the base station. Also, each sensor maintains a variable, called *inversion count*. The inversion count of t he base station is 0. If a sensor chooses one of its low neighbors as its parent, then it sets its inversion count to that of its parent. Otherwise, it sets its inversion count to inversion count of its parent + 1. Furthermore, to deal with the problem of cycles, if the inversion count exceeds a certain threshold ($CMAX$), the sensor removes itself from the tree.

[1]ProSe provides the abstraction which allows a sensor (say, j) to determine whether its neighbor (say, k) is alive or failed. Towards this end, in the input program, sensor j can just access the public variable $up.k$; if $up.k$ is $TRUE$ (respectively, *FALSE*) then k is alive (respectively, failed). ProSe provides implementation of this variable using heartbeat protocol (e.g., [13]).

```
1  program RoutingTreeMaintenance
2  sensor j;
3  const int CMAX;
4  var
5     public int inv.j, d.j;
6     public boolean up.j;
7     private int p.j;
8  begin
9 (d.k < d.j) && (up.k == TRUE) &&
10(inv.k < CMAX) && (inv.k < inv.j)
11    -> p.j = k; inv.j = inv.k;
12| (d.k < d.j) && (up.k == TRUE) &&
13(inv.k+1 < CMAX) && (inv.k+1 < inv.j)
14    -> p.j =k; inv.j = inv.k+1;
15| (p.j != NULL) &&
16  ((up.(p.j) == FALSE) || (inv.(p.j) >= CMAX) ||
17  ((d.(p.j) < dist.j) && (inv.j != inv.(p.j))) ||
18  ((d.(p.j) > dist.j) && (inv.j != inv.(p.j)+1)))
19    -> p.j = NULL; inv.j = CMAX;
20| (p.j == NULL) && (inv.j < CMAX)
21    -> inv.j = CMAX;
22 end
```

Fig. 2. Routing tree maintenance program in shared-memory model

We used ProSe to automatically transform the program into nesC using the TDMA based transformation [10]. ProSe also wires SS-TDMA MAC layer [15] with the generated program. We simulated the generated program (ProSe-RTMP) and MintRoute [16] using TOSSIM [17]. The base station is located at $\langle 0, 0 \rangle$ (i.e., sensor 0) and the inter-sensor separation is 10 ft. We choose 90% as the link reliability. Each sensor executes the write-all action of the program once in every 2 seconds. And, in MintRoute, the sensors exchange routing information every 2 seconds. Once the initial routing tree is constructed, we simultaneously fail some sensors (up to 100 sensors in case of 20x20 network) and measure the convergence time. The results are shown in Figure 3.

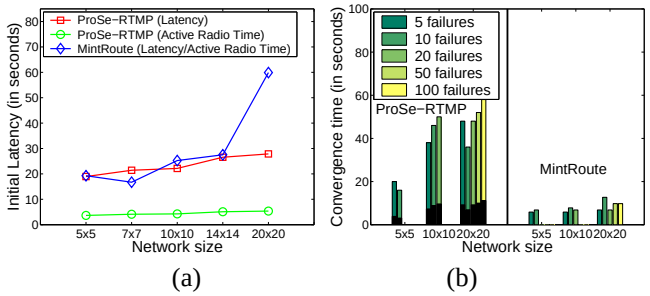

Fig. 3. Simulations results with 90% link reliability: (a) initial latency and (b) convergence time. Note that the black bars in the convergence time graph shows the active radio time during the convergence period.

As observed from Figure 3(b), MintRoute converges to a new routing tree quickly. By contrast, ProSe-RTMP converges within 30-50 seconds. We note that this behavior is *not* because of prototyping with ProSe. Rather, it is because of the nature of the original protocol specified with ProSe. MintRoute is

pessimistic in nature, i.e., it maintains a moving average of link estimates of all active links of a sensor all the time. Hence, when sensors fail, it converges to a new tree quickly. By contrast, RTMP program is *optimistic* in nature, i.e., whenever a sensor chooses one of its neighbors as its parent, it does not change its parent unless the parent has failed or the tree is corrupted. On the other hand, the active radio time during recovery is small with ProSe-RTMP.

2) Intruder Interceptor Program: In this section, we demonstrate the potential of ProSe to program such application level services. We consider the pursuer-evader tracking program proposed in [18]. We specify the evader-centric program from [18] in shared-memory model as shown in Figure 4. In this program, sensors maintain a tracking structure rooted at the evader. The pursuer follows this tracking structure to intercept the evader. Whenever the pursuer arrives at a sensor (say, k), it consults k to determine its next move. Specifically, the pursuer moves to the parent of k. And, since the pursuer is faster than the evader, it eventually intercepts the evader.

```
1  program PursuerEvaderTracking
2  sensor j;
3  var
4    public int distance.j, timeStamp.j, p.j;
5    private boolean isEvaderHere.j;
6  begin
7 (isEvaderHere.j == TRUE)
8    -> p.j = j; distance.j = 0; timeStamp.j = TIME;
9 | (timeStamp.k > timeStamp.j) ||
10   ((timeStamp.k == timeStamp.j) &&
11         (distance.k+1 < distance.j))
12   -> p.j = k; timeStamp.j = timeStamp.k;
13      distance.j = distance.k+1;
14 end
```

Fig. 4. Pursuer-evader tracking program in shared-memory model

We used ProSe to automatically transform the program into nesC using the TDMA based transformation [10] and simulated using TOSSIM [17]. The inter-sensor separation is 10 ft and the TDMA period in SS-TDMA is 0.78 seconds. We choose the link reliability to be 90%. In our simulations, we use a *virtual* pursuer and a *virtual* evader (i.e., modelled using global variables). For more details, we refer the reader to [12]. The ratio of the speed of pursuer movement to evader movement is 1:2. We did two sets of simulations: (1) pursuer at $\langle 0, 0\rangle$ (i.e., sensor 0) and (2) pursuer is initially near the center. In both these scenarios, the evader is initially located at the corner (i.e., at $\langle N-1, N-1\rangle$ on a NxN grid). The simulation results of the generated program are shown in Figure 5. These simulations demonstrate that the program generated by ProSe is competitive to manually designed protocols.

III. Summary

ProSe hides low-level details from the designer. Additionally, as discussed in [12], ProSe: (1) provides abstractions to deal with faults in sensor networks such as state corruption, message loss, and failure of sensors/links, (2) preserves fault-tolerance properties of original guarded commands program during transformation, (3) allows programmers to invoke nesC components/methods in guarded commands, and (4) allows programmers to specify priorities of different actions. We expect ProSe to enable *domain experts* design sensor network protocols rather than *experts in sensor networks*.

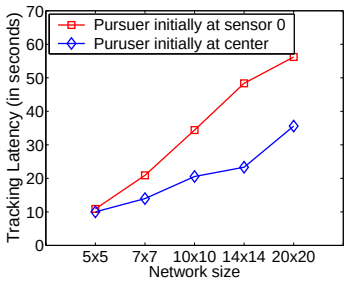

Fig. 5. Tracking latency of the generated program

References

[1] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *Programming Language Design and Implementation*, 2003.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management or, event driven programming is not the opposite of threaded programming," *USENIX Annual Technical Conference*, 2002.

[3] O. Kasten and K. Römer, "Beyond event handlers: Programming sensor networks with attributed state machines," *Information Processing in Sensor Networks*, 2005.

[4] M. Welsh and G. Mainland, "Programming sensor networks using abstract regions," *Networked Systems Design and Implementation*, 2004.

[5] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," *Data Management for Sensor Networks*, 2004.

[6] R. Newton, Arvind, and M. Welsh, "Building up to macroprogramming: An intermediate language for sensor networks," *Information Processing in Sensor Networks*, 2005.

[7] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: A neighborhood abstraction for sensor networks," *Mobile Systems, Applications, and Services*, 2004.

[8] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos," *Distributed Computing in Sensor Systems*, 2005.

[9] K. Whitehouse, F. Zhao, and J. Liu, "Semantic streams: A framework for declarative queries and automatic data interpretation," Microsoft Research, Tech. Rep. MSR-TR-2005-45, April 2005.

[10] S. S. Kulkarni and M. Arumugam, "Transformations for write-all-with-collision model," *Computer Communications*, 2006.

[11] T. Herman, "Models of self-stabilization and sensor networks," *Workshop on Distributed Computing*, 2003.

[12] M. Arumugam, "Rapid prototyping and quick deployment of sensor networks," Ph.D. dissertation, Michigan State University, 2006, available at: http://www.cse.msu.edu/~arumugam/dissertation.pdf.

[13] M. G. Gouda and T. M. McGuire, "Accelerated heartbeat protocols," *International Conference on Distributed Computing Systems*, 1998.

[14] Y.-R. Choi, M. G. Gouda, H. Zhang, and A. Arora, "Stabilization of grid routing in sensor networks," *Aerospace Computing, Information, and Communication*, 2006.

[15] S. S. Kulkarni and M. Arumugam, "SS-TDMA: A self-stabilizing mac for sensor networks," in *Sensor Network Operations*, 2006.

[16] A. Woo and D. Culler, "Taming the challenges of reliable multihop routing in sensor networks," *In Proceedings of the First ACM Conference on Embedded Networked Sensing Systems (SenSys)*, November 2003.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire tinyOS applications," *Embedded Networed Sensor Systems*, 2003.

[18] M. Demirbas, A. Arora, and M. Gouda, "Pursuer-evader tracking in sensor networks," in *Sensor Network Operations*, 2006.