

Programming Model for Supporting Complex Optimizations in Sensor Networks

Raja Jurdak, Cristina Videira Lopes, Pierre Baldi

Donald Bren School of Information and Computer Sciences
California Institute for Telecommunications and Information Technology Cal-(IT)²
University of California, Irvine
{rjurdak, lopes, pfbaldi} @ics.uci.edu

I. PROBLEM

Energy efficiency is a central issue in all sensor network applications. Optimizing energy consumption in a sensor network requires the customization of each node's behavior according to both its local and neighborhood state, which are dynamic and non-uniform. Because of the frequent changes in node states, sensor nodes should autonomously customize their behavior according to their own state.

The sensor node state includes crosscutting aspects, such as:

- 1) **Descendants:** A node's number of descendants in the routing tree rooted at the base station determines its forwarding load. The nodes can use descendant information to optimize the power behavior at each node.
- 2) **Load Balancing:** Energy consumption in sensor networks is inherently nonuniform because data typically flows in specific directions toward one or more data sinks. A load-balancing policy can employ this information to balance energy consumption in the network.
- 3) **Role:** In many networks, nodes may dynamically change their roles during deployment. A policy that considers roles can customize node behavior to reduce energy consumption based on a node's role.

While a policy that considers one of the above aspects can produce significant energy savings, a policy that integrates all of the aspects yields even better energy behavior. Combining the individual policies into an integrated policy is challenging because the different aspects may be at odds. For example, a node's topology position may require the node to sleep more often in order to save energy, whereas the node's role requires that it sleeps less often. This example reveals the need to combine energy-efficient behavior with application requirements. Weaving the integrated policy in the generic sensor application code is another challenge because the programmer must determine which code pieces to insert and where to insert them.

II. MODEL

We propose a model that simplifies the programmer's task in integrating the individual policies and weaving the integrated policy into the sensor code. Figure 1 illustrates the components

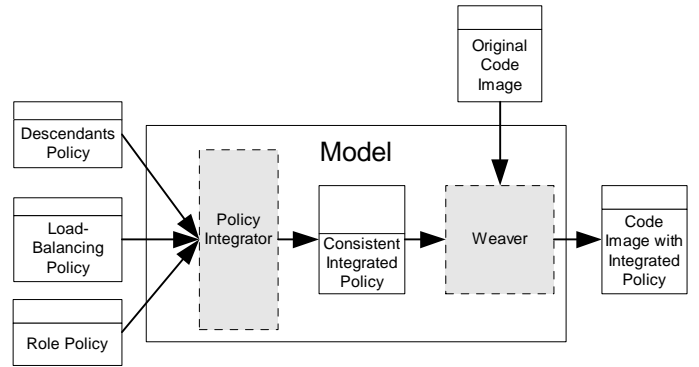


Fig. 1. The Programming Model

of the model, which is inspired by work in Aspect Oriented Programming [1]. We describe the model here through an example.

Our example considers a sensor network with a single data sink. We assume that we have an original code image for the sensor nodes that provides the basic application and communications requirements. To use our model, a programmer first determines the aspects of interest for customizing network behavior. In our example, we consider the three aspects: descendants, load balancing, and role. We consider how to employ these aspects to determine the node's low power listening mode in BMAC [2], which is the main MAC protocol in TinyOS [3].

The second step is for the programmer to specify the policy for each of the aspects. For the descendants aspect, the aim is to select a listening mode that optimizes the node's power consumption for its current forwarding load, which depends directly on its number of descendants in the routing tree. The policy for setting the optimal listening mode for the descendants aspect is:

```
Descendants:
  For every received packet
    If packet forwarded
      packet_counter++
  On timer.fired event
    L=GetBestMode(packet_counter)
    call SetListeningMode(L)
    DeclareListeningMode(L)
```

The function `GetBestMode` determines the optimal listening mode for a given number of packets. `SetListeningMode` is a function provided by the BMAC protocol, and `DeclareListeningMode` informs the node's neighbors of the new listening mode.

Regarding load balancing, a node that is overactive among its neighbors consumes more energy. As a result, the node may change its current listening mode to select a listening mode that saves more energy. The load balancing policy is:

```
Load Balancing:
  On timer.fired event
    C=OwnState/AverageState
    If C>1
      call SetListeningMode(L+floor(C-1))
      DeclareListeningMode(L+floor(C-1))
```

The variable `OwnState` represents the node's own power consumption, and `AverageState` is the average power consumption of its neighbors. If a node decides that its power consumption is considerably higher than average among its neighbors, it switches to a listening mode that save more power. The node also declares its lower power state to its neighbors, which can in turn increase the node's routing cost.

The choice of listening mode also depends on a node's role in the network. For instance, a node that detects a certain event in the environment may choose to dedicate all of its resources for sensing and sending data. This node tries to avoid acting as a forwarder for other nodes and selects its listening mode accordingly. The role-based policy is:

```
Role:
  On Event E
    Role=Reporter
    increase sampling frequency
    call SetListeningMode(M)
    DeclareListeningMode(M)
```

A node that detects event `E` changes its current role to `Reporter` and begins sampling the environment more frequently. The node then sets its listening mode to `M`, which enables the node to listen infrequently and save power. The node also declares its listening mode so that neighbors avoid using it as a forwarder.

The three policies above all share a common goal, which is the optimization of energy consumption under certain conditions. However, combining the policies into an integrated policy is challenging because certain network conditions may cause each policy to select a different listening mode. A consistent integrated policy must combine all policies while preserving the intended behavior of each policy. In our model, the policy integrator is responsible for creating an integrated policy. The policy integrator determines the points at which the individual policies can be linked and merges the policies into a integrated policy. The integrated policy becomes:

```
Integrated Policy:
For every received packet
  If packet forwarded
    packet_counter++
On timer.fired event
  If Role!=Reporter
    L=GetBestMode(packet_counter)
    C=OwnState/AverageState
    If C>1
      call SetListeningMode(L+floor(C-1))
      DeclareListeningMode(L+floor(C-1))
    Else
      call SetListeningMode(L)
      DeclareListeningMode(L)
  Else
    call SetListeningMode(M)
    DeclareListeningMode(M)
On Event E
  Role=Reporter
  Increase sampling frequency
  call SetListeningMode(M)
  DeclareListeningMode(M)
```

Let's consider how the integrator merges the logic for the periodic `timer.fired` event into consistent integrated logic. The integrator can first identify operations that handle common variables and calls to the same functions, which represent the critical join points for policies. In this example, we only have common function calls, namely `SetListeningMode` and `DeclareListeningMode`. The integrator can treat all logic between the join points as independent and simply append the logic of different policies in between join points. In the load balancing policy, calls to the common functions are located in a conditional statement, whereas the descendants policy calls the functions unconditionally. The integrator can infer that the integrated policy should include the common function calls inside a conditional statement. Consistency imposes that the integrated policy calls common functions only once per invocation, so the integrator places the calls from the descendants policy in an `else` statement to be executed only if the calls for load balancing policy are not. Similar observations enable the integrator to produce the fully integrated policy.

The model can easily map the policy logic into components in NesC code, so the weaver can place that logic in the correct NesC components. The weaver also determines what additional background processing is required to enforce the integrated policy and where to make the corresponding code changes. Additional background processing typically includes adding necessary fields in packets and keeping track of relevant neighborhood states. The weaver yields the final sensor code, which consistently and efficiently merges the functionality of the original code with the integrated policy.

REFERENCES

- [1] G. Kiczales, J. Lamping, A. Mendhekar et al. Aspect-Oriented Programming. In proc. (*ECOOP'97*), Springer-Verlag LNCS n.1241, 1997.
- [2] J. Pollastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. Proceedings of *ACM SenSys*, 2004.
- [3] Tiny Operating System. UC Berkeley. available: <http://tinysos.net>.