

PRO 06 Separation of Concerns for Sensor Network Software

PRO 06.1 People

- Principal Investigators: Todd Millstein
- Faculty: Ramesh Govindan, Computer Science, USC. Todd Millstein, UCLA.

PRO 06.2 Overview

Sensor network applications must cope with a variety of concerns, including severe resource and energy constraints, consistency and synchronization among nodes, and node failures. Current sensor network platforms therefore allow the application programmer to closely monitor and directly control the handling of various resources. While this design enables programmers to obtain acceptable application performance, it forces the code for an application's functionality to be tangled with the code for handling non-functional concerns like resource constraints. Such tangling makes sensor network programming complicated, tedious, and prone to errors. It also makes sensor network components hard to reuse across applications, since the application logic is inextricably linked with many other concerns.

Similarly, tangled concerns prevent programs from being easily adapted to changes in the environment, new performance requirements, or new functional requirements. Most sensor network applications make tradeoffs among various concerns, in order to obtain acceptable application performance while satisfying resource constraints and maintaining energy efficiency. Since the performance of sensor network applications can be quite sensitive to the hardware platform and environment, the application designer may have to vary the tradeoffs and decisions she makes for various concerns depending on the particular deployment scenario. Doing this while the concerns are entwined with the functionality of the application can be quite difficult, resulting in unintended changes in functionality and other errors. Since sensor networks are known to be notoriously hard to debug (various debugging systems such as Sympathy, Nucleus, Clairvoyant, and Hermes are testament to this), tangled concerns can thus result in a lot of wasted programmer effort and time.

PRO 06.3 Approach

Separating various non-functional concerns from application logic, while still retaining programmer control over their handling, solves a number of the above problems. It promotes code reuse, by allowing generalized solutions to be provided for individual concerns. It also decouples application functionality from various non-functional concerns, improving visibility into the application logic and making it easier to analyze, while exposing tradeoffs among concerns which can be tweaked independently from the application functionality. Thus, a separation of non-functional concerns from application logic is quite desirable.

Some concerns like communication etc., can be easily separated into logical sections, using modular programming techniques and by designing modular architectures for sensor network systems. This insight has been used in the design of TinyOS to allow for separation of simple concerns into modules. Also, researchers have designed modular architectures atop TinyOS, which separate various communication and link-layer related concerns, as well as device-level power management concerns from the rest of the system.

For a number of concerns, however, it is quite difficult to cleanly separate them from the rest of the program, in both design and implementation, using the above-mentioned techniques. The code handling them is scattered throughout the system, and/or tangled with various other concerns. These concerns are known as cross-cutting concerns. Sensor network applications have their share of cross-cutting concerns which are hard to separate from application logic. Some examples of such concerns are fault tolerance, energy management, etc.

Researchers have identified a few high-level/cross-cutting concerns like energy management, heterogeneity, and failure recovery, and worked on individual solutions for separating them from application functionality. For example, language based solutions have been proposed to separate energy management from the application logic. Solutions have also been proposed to handle heterogeneity and role assignment and failure recovery. These

solutions are very specific, applicable only to the particular concern that they try to separate, and hence not generalizable to other concerns.

As the sensor networking hardware and applications are getting more advanced, researchers are aiming for not only energy efficiency, but also improved performance and robustness of their programs. Consequently, a number of new high-level, cross-cutting concerns are rising into prominence. A number of applications now list as one of their requirements concerns such as fault-tolerance, timing constraints, and reliability of communication. Currently, there is no one single solution which separates and handles all these concerns.

PRO 06.4 System Description

We propose the use of annotations to separate various high-level concerns from the application logic. The idea is that the application designers can annotate their application logic with various annotations representing how each concern should be handled. These annotations may be parameterized, that is, they permit a degree of control over their function. Also, the set of annotations may be extended by the programmer if they want to address a concern in a fashion not provided by any of the existing annotations.

We are leveraging our earlier programming platform for sensor networks, Pleiades, as a testbed for our ideas on separation of concerns. Pleiades is an example of a “macroprogramming” language for sensor networks. In this style, a sensor network is programmed centrally, with the compiler automatically partitioning the application to run on the individual nodes of the network.

A macroprogramming language is a natural starting point for support of separation of concerns, since it already raises the level of abstraction over traditional node-level programming. Annotations for separation of concerns can leverage this higher level of abstraction for increased expressiveness and usability.

Pleiades currently provides a default handling for all non-functional concerns, as implemented by the Pleiades compiler and runtime system. For example, the Pleiades compiler and runtime system coordinate to determine a strategy for implementing an application’s functionality across a network in a manner that minimizes communication costs. The Pleiades language also automatically ensures serializability for its concurrent for loop, thereby guaranteeing a strict form of consistency across nodes. Our aim is to design annotations that can augment Pleiades to allow programmer flexibility and control in the handling of these and other concerns.

PRO 06.5 Accomplishments

This year we have focused on an important non-functional concern for sensor networks: fault tolerance. Previously, the Pleiades implementation was vulnerable to node failures, with even a single node failure able to cause the application as a whole to fail. We have augmented the Pleiades system with a strategy for automatically detecting and recovering from failures. Further, we have implemented a declarative set of program annotations that allows both failure detection and recovery to be controlled by the application programmer, while keeping this concern completely separate from the main application logic.

A node is considered to have failed if a request to the node times out. For example, a failure could be detected when the Pleiades runtime attempts to obtain the value of a variable stored on a particular node by sending the node a message, but the node does not respond. Programmers can configure this failure detection algorithm by supplying a timeout period via an annotation.

Pleiades also provides annotations to allow the user to specify desired recovery actions to take whenever a particular type of failure occurs. For example, variables may be annotated to specify default values to be used in case the variable cannot be accessed due to node failure. In a car-parking application that we have implemented, each node maintains a Boolean variable `isfree` indicating if the node’s associated parking spot is available. Setting the variable’s default value to false ensures that if a node fails, the execution continues with the assumption that the failed node does not have a free parking spot, and hence gracefully deals with this failure.

Finally, we have incorporated special annotations for handling failures during a Pleiades `cfloop`, which concurrently executes each iteration at a different node in the network. If a node executing a `cfloop` iteration fails,

this is detected by the Pleiades runtime system. The Pleiades language allows the user to annotate cfor loops with a condition on how many failed iterations can be tolerated before having to consider the entire cfor as being failed. For example, an annotation could indicate that failures of up to 50% of the nodes executing a cfor iteration can be tolerated, but beyond that threshold the entire loop should be aborted and considered failed.

PRO 06.6 Future Directions

Our initial experience with declarative annotations for detecting and recovering from failures has been positive. However, failure is just one of a number of non-functional concerns of importance to sensor network applications. In the next year, we plan to leverage our experience with failures to design an annotation language that is declarative and easy to use while allowing 7.4 for the specification of a variety of different concerns. This is a daunting task with a number of important issues that must be addressed:

Determining exactly how much control to give to the programmer over the handling of various concerns: Should we provide certain concern-specific primitives that the programmer can use to construct their own mechanisms, should we only allow for simple tuning of the mechanisms using parameters, or should we give the programmer free reign?

How extensible should the annotation language be?: We already recognize that extensibility is an important goal, and ideally, our framework should be completely extensible, in that new concerns should be easily incorporated and handled. However, we worry that letting the system be so extensible would make it complicated and unwieldy to use.

Design of a general mechanism to describe tradeoffs: We would like to have a simple, and general mechanism by which programmers can define their performance requirements for the application, and also describe the tradeoffs among various concerns.