Poster Abstract: Balancing Visibility and Resource Consumption for Long-Term Monitoring of Sensornets

Junyan Ma†*
alexmajy@acm.org

Kay Römer‡*
roemer@inf.ethz.ch

†School of Computer Science, Northwestern Polytechnical University, China
*Institute for Pervasive Computing, ETH Zurich, Switzerland
‡Institute of Computer Engineering, University of Lübeck, Germany

Abstract

Limited visibility of node states makes debugging deployed sensor networks very difficult. Higher visibility usually implies more resource consumption. As sensor networks are resource-constrained and need to operate unattended for long periods, a balance between a sufficient level of visibility and a tolerable consumption of resources needs to be found so that enough evidence can be collected to analyze the causes of observed problems. We propose a mechanism called *visibility levels (vLevels)* to manage the trade-off between visibility and resource consumption. Preliminary experimental results show the feasibility of vLevels.

1 Introduction

Problems in deployed sensor networks are not only common, but very hard to debug. A key requirement for debugging deployed sensor networks is visibility of the system state, i.e., the ability of an engineer to observe the internal program state of the sensor nodes. As visibility requires communication to expose internal node state to an external observer, higher visibility results in higher resource consumption. Since resources are scarce in sensor networks, a trade-off between a sufficient level of visibility and a tolerable consumption of resources needs to be found. This is especially true for situations where the system state needs to be observed over a long period of time in order to collect enough evidence to make an informed decision about the causes of observed problems. Inspired by Energy Levels [2], a tool called *vLevels* is proposed to provide a tuning knob to the user to specify a resource budget such that best possible visibility is achieved while not exceeding this budget.

2 Design

vLevels offers three core abstractions: *visible objects*, *visibility levels* and *observation scheme*. A *scheduler* implements the allocation of resources to visible objects.

2.1 Abstractions in vLevels

Visible objects are user-defined slices of system state. By using a simple declarative language, similar to LIS [3], the user can specify which part of the program state is of interest to him. Essentially, a visible object specifies an event (such as changes of variable values or invocations of certain functions) and a set of program variables, with the semantics that whenever the event occurs a snapshot of the variables should be created and logged together with a timestamp of the event occurrence.

Table 1. Example visibility specification.

```
1 fvar app.c:func1:light
2 vlevels lreading for light
3 {
4   log light
5
6   filter light > 200
7   log light
8
9   log
10}
```

```
1 oscheme algo
2 {
3    budget = 60 BPS
4    light@levels = lreading
5    light@priority = 2
6    light@rpolicy = DEE
7    elect@levels = etimeout
8    elect@priority = 1
9    elect@rpolicy = DEE
10}
```

For each visible object, the user can define an ordered set of **visibility levels**. Each visibility level essentially constitutes a lossy compression function that is defined in a declarative manner using a number of basic operators. Every level implements a specific trade-off between accuracy of the snapshot of a visible object on the one hand, and size of the snapshot and therefore resource consumption on the other hand. Moreover, the visibility levels of a visible object form a pipeline: The output of level k forms the input of level $k-1$. By definition, level 0 produces empty output and thus gives zero visibility at zero cost. That is, both visibility and resource consumption monotonically increase with increasing level numbers.

An **observation scheme** defines a resource budget (communication bandwidth, or amount of storage) and policies how to prioritize visible objects among each other, and how to prioritize different snapshots of a single visible object among each other, such that a scheduler can automatically select a visibility level for each visible object so as to maximize visibility while not exceeding the resource budget.

Let us consider the example in Table 1. On the left, the keyword `fvar` in line 1 defines a local variable (`light`) of a given function (`func1`) in a given source file (`app.c`) as a visible object, the event being any assignment of a value to that variable. On the left, lines 2 to 10 contain the definition of a set `lreading` of three visibility levels separated by blank lines, for the previously declared visible object (`light`). Level 3 (line 4) just outputs the raw snapshot of

SenSys'10, November 3–5, 2010, Zurich, Switzerland.
ACM 978-1-4503-0344-6/10/11

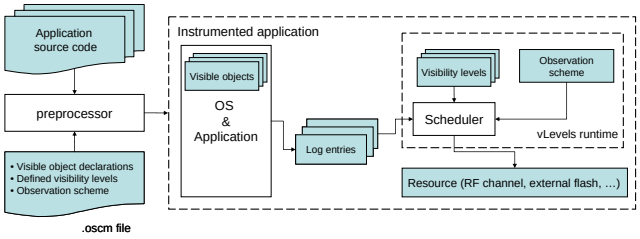

Figure 1. System architecture of vLevels.

the variable `light` using the `log` operator. Level 2 (line 6 to 7) uses the `filter` compression operator to ignore all snapshots ≤ 200. Level 3 (line 9) takes the output of level 2 as input and creates an empty log entry. As log entries do also contain a timestamp, this is an indicator of *when* the variable has been assigned values > 200. As we can observe in this example, the visibility of the `light` variable as well as the size of the generated log entries are monotonically increasing with increasing level number. The right side of Table 1 defines an observation scheme named `algo`. The `budget` keyword specifies a bandwidth budget of 60 bytes per second (BPS). The `@levels` attribute specifies a set of visibility levels for a visible object and the `@priority` attribute defines the relative importance of the visible objects. Finally, the `@rpolicy` attribute defines the policy how to prioritize different snapshots of the same visible object among each other, such as `DEE` that stands for Drop Earliest Entry.

2.2 Scheduler

The scheduler is the component that dynamically selects visibility levels for visible objects such that the given resource budget is not exceeded. The key idea is that the scheduler maintains a buffer of a fixed size and enters new log entries at the end of the buffer. If the remaining space in the buffer is not sufficient to hold the new entry, then one or more existing log entries in the buffer are selected for reduction of their visibility level according to the priorities and policies specified in the observation scheme.

3 Implementation

Our prototype implementation of vLevels on the Contiki operating system consists of two main parts: a preprocessor and a runtime system.

Fig. 1 depicts the architecture of vLevels. To work with vLevels, the user creates an additional *.oscm* input file that contains the specification of visible objects, visibility levels, and observation schemes. Then, the C source code and the .oscm file are fed into the *preprocessor* that performs code instrumentation and generation. The resulting source code is then compiled, linked with the *runtime* library that consists of the scheduler and a module for storage or wireless transmission of buffers containing log entries, and uploaded to the sensor nodes. When the application is executing on the sensor node, the instrumented code generates snapshots of the state of visible objects. These timestamped snapshots are then passed to the scheduler that dynamically selects an appropriate visibility level for each snapshot such that the resource budget specified in the observation scheme is not exceeded. The resulting compressed snapshots are then either stored in (flash) memory for post-mortem analysis or sent over the wireless channel for online analysis.

4 Preliminary Results

To verify the feasibility of our design, we conduct a preliminary experiment on applying vLevels to a target tracking application. We choose Tmote sky as our sensor node hardware platform. Four visible objects with appropriate visibility levels are declared to observe the execution of the application. The observation scheme specifies a bandwidth budget of 4 bytes/s.

The size of the original application is 29510 bytes (ROM) and 5956 bytes (RAM). Instrumenting it with vLevels and a logging buffer size of 50 bytes, RAM consumption is increased by 210 bytes or 2% of the total RAM of the Tmote Sky, and 4298 bytes of ROM or 8% of the total ROM of the Tmote Sky. We also studied the runtime overhead in terms of processor cycles by running the application in the COOJA/MSPSim cycle-accurate simulator [1]. We find a total overhead of 390942 cycles or 0.03% for the studied application. During a 300 seconds experiment where the target appears at 50 seconds, remains static and disappears at 250 seconds, the resulting throttled bandwidth is between 3.24 bytes/s for a logging buffer size of 24 bytes and 3.94 bytes/s for a logging buffer size of 12 bytes.

5 Conclusions

Finding the right balance between sufficient visibility and tolerable resource consumption is crucial for long-term monitoring of sensor networks. The major contribution of this paper is the presentation of a novel framework that allows the user to specify a resource budget and the runtime provides best possible visibility into the system state while not exceeding the resource budget.

6 Acknowledgments

This work has been partially supported by the Swiss National Science Foundation (NCCR-MICS, 5005-67322), the European Commission (CONET, FP7-2007-2-224053) and the National Key Technology R&D Program of China (2007BAD79B00).

7 References

[1] J. Eriksson et al. Cooja/mspsim: interoperability testing for wireless sensor networks. In *Simutools 2009*, pages 1–7, ICST, Brussels, Belgium, Belgium, 2009. ICST.

[2] A. Lachenmann et al. Meeting lifetime goals with energy levels. In *SenSys 2007*, pages 131–144. ACM, 2007.

[3] R. S. Shea et al. Lis is more: Improved diagnostic logging in sensor networks with log instrumentation specifications. Technical Report TR-UCLA-NESL-200906-01, June 2009.