Visibility Levels: Managing the Tradeoff between Visibility and Resource Consumption*

Junyan Ma[1,2] and Kay Römer[2,3]

[1] School of Computer Science, Northwestern Polytechnical University, China
[2] Institute for Pervasive Computing, ETH Zurich, Switzerland
[3] Institute of Computer Engineering, University of Lübeck, Germany

Abstract. Pre-deployment tests of sensor networks in indoor testbeds can only deliver a very approximate view of the correctness and performance of a deployed sensor network and it is therefore common that after deployment problems and failures occur that could not be observed during pre-deployment tests. Finding and fixing such problems requires visibility of the system state, such that an engineer can identify causes of misbehavior. Unfortunately, exposing the internal state of sensor nodes requires resources such as communication bandwidth and energy: the better visibility of system state is required, the more resources are needed to extract that state from the sensor network. In this paper we propose a concept and tool that give the user explicit control over this tradeoff. Essentially, the user can specify a resource budget and our tool strives to provide best possible visibility while not exceeding the resource budget. We present the design of our vLevels framework and report the results of a case study demonstrating that the overhead of our approach is small and that visibility is automatically adjusted to meet the specified resource budget.

1 Introduction

Being deeply embedded into the real world, the function of sensor networks is heavily affected by their interaction with the environment. Therefore, pre-deployment tests in testbeds can only deliver a very approximate view of the correctness and performance of a deployed sensor network and it is therefore common that after deployment problems and failures occur that could not be observed during pre-deployment tests. Finding and fixing such problems is difficult due to limited access to the deployed network – both physically and due to constrained resources.

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. Unfortunately, there is a fundamental tradeoff between visibility and resource consumption. As visibility requires communication to expose internal node states to an external observer, increasing visibility requires more resources. Since resources are

* 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).

P.J. Marron et al. (Eds.): REALWSN 2010, LNCS 6511, pp. 49–61, 2010.

scarce in sensor networks, a balance 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.

While there exist a number of tools to give an observer visibility into internal node states, they do not offer a turning knob to the user to select a reasonable balance between visibility and resource consumption. With these existing tools, the user can define which states should be visible, and the resource consumption is implied by these requirements. However, it is very difficult for the user to predict the resulting resource consumption and hence almost impossible to exert fine-grained control over the permissible amount of resource consumption.

The goal of our work is to provide the user with an explicit turning knob to balance visibility and resource consumption. With our approach, the user can specify both visibility requirements and a resource budget and our system will provide best possible visibility while not exceeding the given resource budget. Here, visibility means the creation of traces of the system state that are either logged into memory for later post-mortem analysis, or transmitted over the wireless channel for online inspection. The resource budget is therefore defined in terms of available storage space or communication bandwidth. Our approach is embodied in a software framework called *vLevels*. The paper continues with the design of vLevels in Sect. 2, implementation aspects in Sect. 3, a case study in Sect. 4, and with a summary of related work in Sect. 5. Finally, we provide our conclusions in Sect. 6.

2 Design

vLevels offers visibility into the system state by creating snapshots of user-defined slices of the system state. The key innovation in vLevels is that it provides a turning knob for the user to specify a resource budget such that best possible visibility is offered while not exceeding this budget.

To realize this turning knob, vLevels offers three core abstractions: *visible objects* are slices of system state that should be made visible to an observer. When a user-define event occurs (such as changes of variable values or invocations of certain functions), a snapshot of the state of the visible object is logged. For each visible object, the user can define an ordered set of *visibility levels*. A visibility level is a user-defined lossy compression function that compresses a snapshot of the state of a visible object. Higher visibility levels result in more accurate snapshots which also consume more resources. Lower visibility levels result in less accurate snapshots which consume less resources. Finally, an *observation scheme* defines a resource budget and assigns priorities to visible objects, 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. We continue with an overview of the architecture of vLevels, before discussing the components of the architecture in detail.

2.1 System Architecture

We assume a traditional node-centric programming model, where developers write code in an imperative programming language such as C that is compiled, uploaded, and

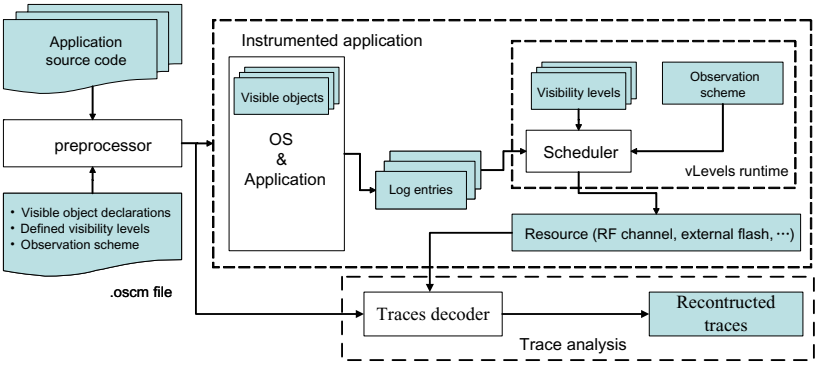

Fig. 1. System architecture of vLevels

executed at the sensor nodes. With vLevels, the user creates an additional *.oscm* input file that contains the specification of visible objects, visibility levels, and observation schemes. As illustrated in Fig. 1, both the source code and the .oscm file form the input to a preprocessor that modifies the source code such that snapshots of the state of visible objects are generated when certain events occur. The resulting source code is then compiled, linked with a vLevels runtime library, 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 a 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 that constitute the traces of the system state are then either stored in (flash) memory or sent over the wireless channel. Using the compressed traces (that include timestamps and the chosen visibility levels) downloaded from the flash memory or received over the wireless channel, the original uncompressed traces of visible objects can be approximately reconstructed and analyzed by the user. The lower the visibility levels used for compression, the less accurate the reconstructed states will be. For the reconstruction, the compression schemes defined by the visibility levels are compiled by the preprocessor into appropriate decompression code that can be executed on the user's computer.

As the above discussion indicates, vLevels builds upon a number of fundamental services such as storage management (e.g., Contiki's Coffee filesystem [15]), timestamping and synchronization services as well as data collection (e.g., as described in [12]). Due to space constraint we focus on the innovative aspects of vLevels in this paper, rather than re-iterating those fundamental and well-understood services.

2.2 Visible Objects

The complete state of a program is typically too large to make it visible in its entirety. Therefore, a user has to specify which part of the program state is of interest to him. In vLevels, the *visible object* abstraction allows a user to specify the interesting state. Essentially, a visible object specifies an event 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.

vLevels offers a simple declarative language to define visible objects. We opted to separate the specification of visible objects from the program source code to avoid mixing the two different aspects of program logic and visibility, which is inspired by aspect oriented programming [6] in general, and Declarative Tracepoints [2] and LIS [14] in particular.

In the following we illustrate different types of visible objects in vLevels by example using our specification language. In general, a visible object is specified by a keyword that defines the event which is followed by parameters detailing the event and variables as in the following examples:

```
1 var tracking.c::m_state
2 fvar tracking.c:sample_light_process:light
3 fhdr reports_mngmt.c:leader_report_update:reading x y
4 tpoint tracking.c:tp_leader_cycle:report_nr
5 var estate as example.c::m_state
```

The `var` keyword in line 1 defines a single global variable as a visible object, the event being any assignment of a different value to that variable. The parameters specify the name of the source file (`tracking.c`) and the name of the variable (`m_state`). The `fvar` keyword in line 2 is similar, but refers to a local variable (`light`) of a given function (`sample_light_process`) in the given source file.

Keywords `fhdr` (function header) in line 3 and `fftr` (function footer) also refer to local variables of a function including in particular the function parameters, but the triggering event is the invocation of the function (`fhdr`) or the return from the function (`fftr`). In the example in line 3 the parameters `reading`, `x`, and `y` of function `leader_report_update` are logged just before the first instruction of the function is executed.

The keyword `tpoint` (trace point) in line 4 generalizes this concept, where the triggering event is when control flow reaches a given point in the program. As the specification of line numbers is very error prone, we opted for inserting a marker comment into the source code, such that the snapshot of a given set of variables is generated whenever the control flow reaches this marker in the code. The code below shows such a marker comment, which has to follow the format `@visible tpoint` followed by the name of the tracepoint. This name (`tp_leader_cycle`) is also given in the specification of the visible object, followed by the names of the variables.

```
...
  target_pos_estimation(&target);
  /* @visible tpoint tp_leader_cycle */
  member_reports_clear();
...
```

For keywords `var` and `fvar`, variable names are used as their corresponding visible object names by default. Likewise, function names and tracepoint names are used as names of visible objects defined with `fhdr`, `fftr` and `tpoint`. The `as` keyword in line 5 is used to rename a visible object when there is a conflict.

2.3 Visibility Levels

Visibility levels are user-defined compression schemes to reduce the size of traces generated by visible objects. In particular, a set of visibility levels can be defined for each

visible object. The levels in each set are ordered and numbered with small integers. Every level implements a specific tradeoff 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. Assuming there are N levels, then level N gives maximum visibility but also maximum resource consumption. 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. We will explain later in the paper how the visibility levels of a set of visible objects can be automatically selected such that a given resource budget is not exceeded.

Each visibility level essentially constitutes a lossy compression function that is defined by the user in a declarative manner using a number of basic operators. Moreover, the visibility levels of a visible object form a pipeline: The output of level k forms the input of level $k-1$. Level N is typically the raw snapshot of a visible object. The rationale for this design will become clear in Sect. 2.5, where we describe the scheduler that selects the visibility levels. The basic operation of the scheduler is to incrementally decrease the visibility levels of a snapshot by one. The pipelined design of visibility levels makes this a very efficient operation as the raw snapshot does not have to be saved in order to change the visibility level later.

The left part in Table 1 shows the definition of a set of three visibility levels for a program variable `light` that has been declared a visible object as described in Sect. 2.2 (line 2 in the example there). The example declares three visibility levels, the declarations of which are separated by blank lines. Each visibility level consists of zero or more compression operations (one per line) followed by a single output operation `log` to produce the output.

Visibility level 3 is declared first in line 3, followed by level 2 in lines 5 and 6, and finally level 1 in line 8. Level 3 just outputs the raw snapshot of variable `light` using the `log` operator. Level 2 uses the `filter` compression operator to ignore all snapshots where the `light` value is ≤ 200. The remaining `light` values are output with `log`. Level three takes the output of level 2 as input, but instead of outputting the light value, it just creates an empty log entry using `log` without parameters whenever a new value > 200 is assigned to variable `light`. As log entries do also contain a timestamp, this is an indicator of *when* the variable has been assigned new values > 200, but not which values. 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.

Besides `log` and `filter`, vLevels also provides `remapper` and `bits`. Operator `bits` (bits selector) selects a subset of the bits of its input using a bit mask. `bits` is often used to deal with network messages to extract the relevant protocol fields. Operator `remapper` remaps a set of integer values to a new set of integer values. Essentially, this operator reduces the precision of a scalar value to a smaller number of bits.

We conclude this section with a discussion of the design rationale behind visibility levels. Essentially, with vLevels a user can define custom lossy compression schemes, allowing him to exploit his domain knowledge about what is important state (and should not be lost during compression) and what is not. This is especially important for the compression of systems logs containing very different data types (e.g., outputs of

Table 1. Visibility levels and observation scheme

Visibility Levels

```
1 vlevels lreading for light
2 {
3   log light
4
5   filter light > 200
6   log light
7
8   log
9 }
```

Observation Scheme

```
1 oscheme leader_algorithm
2 {
3    budget = 60 BPS
4
5    light@levels = lreading
6    light@priority = 2
7    light@rpolicy = DEE
8
9    election_timeout@levels = electmout
10   election_timeout@priority = 1
11   election_timeout@rpolicy = DEE
12 }
```

different sensor types, program state variables). An alternative design could use general purpose compression algorithms. However, they are often optimized for specific data types and do not allow to use to incorporate domain knowledge during compression.

2.4 Observation Schemes

An observation scheme specifies how to select the visibility levels of a set of visible objects such that a certain resource budget is not exceeded. In addition to the resource budget itself, the observation scheme also defines policies how to prioritize visible objects among each other, and how to prioritize different snapshots of a single visible object among each other. In the case that the resource budget is not sufficient to log all snapshots of all visible objects at maximum visibility level, these policies control the selection of visible objects whose visibility level needs to be lowered. Let us consider the observation scheme named `leader_algorithm` in the right part of Table 1.

The `budget` keyword is used to define a bandwidth budget of 60 bytes per second (BPS), meaning that the log data is transmitted online over a communication channel and the log data bandwidth should not exceed 60 BPS. Based on the application requirements, the user decides how much resources can be spent for monitoring and selects the budget appropriately. The budget can also be changed later during runtime. As the radio is the dominating energy sink, with a simple calibration step it is possible to map bandwidth to approximate energy consumption, such that instead of specifying a bandwidth budget, one can also specify an energy budget. The system also supports storage of log data in flash memory for later offline analysis. In the latter case, a storage budget is specified in units of bytes (B), kilobytes (KB), or megabytes (MB).

The example observation scheme considers two visible objects: `light` and `election_timeout`. By assigning the name of a visibility level set to the `@levels` attribute, a visibility levels set can be selected for a visible object as in lines 5 and 9. The `@priority` attribute (lines 6 and 10) defines the relative importance of the visible objects, where smaller values mean more importance. If there are not enough resources to log both objects at maximum visibility level, then preference (i.e., higher level) will be given to the object with the smallest priority (here, the `election_timeout` object).

However, the visibility level can also change dynamically among different snapshots of the same visible object. The `@rpolicy` attribute (lines 7 and 11) defines the policy how to prioritize different snapshots of the same visible object among each other. For

example, the DEE (Drop Earliest Entry) default policy specifies that priority should be given to the latest snapshots, i.e., the earliest snapshot should be dropped first if the budget is not sufficient. Other supported policies are DLE (Drop Latest Entry) and MMTIU (Minimize Maximum Time Interval between Updates). The latter policy drops the snapshot that results in the minimum increase of time gap between successive snapshots in the trace, which is suitable for signal reconstruction if the visible object represents the output of a sensor.

2.5 Scheduler

The scheduler is the component in our system that dynamically selects visibility levels for visible objects such that the given bandwidth/energy or storage budget is not exceeded. It should be noted that in our system the budget is considered a user-defined constant and the scheduler assumes this budget is always available during runtime. 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.

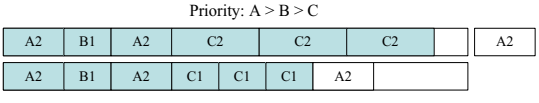

Fig. 2. Examples of the scheduler algorithm

In case of logging into (flash) memory for later offline analysis, this buffer equals the storage space in the memory. An efficient file system such as Coffee [15] is required to manage access to the flash. In case of online monitoring, the contents of the buffer are sent over the communication channel at regular intervals, such that this interval equals the buffer size divided by the bandwidth budget. As transmission of the buffer contents is not instantaneous, a second buffer is used. While one of the buffers is being filled with log entries, the contents of the other buffer are transmitted in the background.

Both in the online and offline modes the basic operation of the scheduler is as follows. When appending a new log entry to the buffer, it is initially inserted with the current visibility level of the visible object. If the available space in the buffer is not sufficient to hold the new entry, then the visibility level of the object with the lowest priority is reduced by one until there is enough space in the buffer to hold the new entry. The core operation of the scheduler is hence to reduce the visibility level of a log entry in the buffer by one until it eventually reaches zero, i.e., the log entry disappears. This is also the reason for the pipelined design of the visibility levels as described in Sect. 2.3, since reducing the visibility level of an entry in the buffer from k to $k-1$ is then efficiently implemented by applying the compression function of level $k-1$ to the entry. If the lowest priority object is the object associated with the new log entry and its level is already 1, then a log entry is selected to be replaced with the new one according to the replacement policy. Figure 2 gives an example of how the scheduler works. The notation A2 stands for a log entry of visible object A with visibility level 2. As shown

in the figure, a new entry A is added but there is not enough space in the buffer. As C is the visible object with the lowest priority, the visibility level of all C entries is reduced from 2 to 1.

3 Implementation

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

The *preprocessor* reads the C source files and the `.oscm` file containing the specification of visible objects, visibility levels, and observation schemes. In particular, the preprocessor assigns a unique identifier to every visible object to identify log entries; source code is instrumented to log snapshots of visible objects; compression functions are generated for visibility levels; parameters from the observation schemes are extracted and passed to the scheduler; and a trace decoder is also generated for reconstructing traces from the collected logs. The instrumentation part is implemented using CIL (C Intermediate Language) [9]. Operating on the intermediate representation of C programs generated by CIL, code analysis and transformation are performed, such as code injection after an assignment to a visible variable.

The *runtime* mainly consists of the scheduler and a module for storage or wireless transmission of buffers containing log entries. The runtime maintains a separate thread for sending off buffers with log entries in the background.

In our current implementation, changing .oscm file requires to preprocess and compile the code and to upload the new image to the sensor nodes. However, through binary instrumentation techniques [2] and run-time dynamic linking [3], it would also be possible to insert new visible objects, update modules and load new modules into an executing program without losing its state. We leave this aspect for future work.

4 Case Study

To verify the feasibility of our design, we conduct a preliminary experiment on applying vLevels to a target tracking application similar to EnviroTrack [1]. We investigate memory overhead, runtime overhead, as well as the impact of buffer size on observation accuracy and bandwidth throttling. We choose Tmote sky as our sensor node hardware platform and Contiki [4] for the operating system running on the nodes. The experiment is carried out using the cycle-accurate COOJA/MSPSim simulator [5].

4.1 Tracking Application

We consider a simplified single-target tracking application where sensor nodes can sense the proximity of the target (e.g., using an IR light sensor to detect the presence of living beings). If the sensor reading is above threshold the target is considered detected. One of the nodes close to the target is elected as the leader and all detecting nodes send messages with their sensor values and locations to the leader which computes the target location and notifies a sink. When the target moves away from the the leader, the leader role is handed over to a closer node. The tracker is implemented as a state machine with

states *idle* (no target detected), *leader*, *candidate* (target detected but not a neighbor of the leader), *member* (target detected and neighbor of a leader), *sentry* (no target detected but neighbor of leader), or *temporary* (a leader that lost the target). A candidate turns into leader if it did not receive an announcement from another leader during a certain timeout.

4.2 Visibility Specification

In our experiment, four visible objects are declared to observe the execution of the application. The first visible object is the state `m_state` of the tracker with three visibility levels: level 3 (the original value of every assignment to the variable), level 2 (one bit indicating if the state is unstable (i.e., *candidate* or *temporary*) or stable (i.e., remaining states)), level 1 (empty log entries indicating the assignment of the state variable). The second visible object is the local variable `light` in function `sample_light_process`, holding the most recent sensor reading. There are three visibility levels: level 3 (original variable values), level 2 (values greater than the detection threshold), level 1 (empty log entries indicating assignment of values greater than the threshold). The third visible object is the invocation of `election_timeout`, the timeout callback function for leader election. There is only one visibility level (empty log entries indicating the invocation of the function). The last visible object is `tp_leader_cycle`, a tracepoint in the target estimation function of the leader creating snapshots of four variables: the number of report messages received from members, the average sensor reading computed by leader, and the estimated (x, y) location of the target. There are four visibility levels: level 4 (values of all variables), level 3 (number of reports, the most significant 4 bits of the target position (x, y), the most significant 8 bits of the average sensor reading), level 2 (number of reports), level 1 (empty log entry indicating the hit of the tracepoint).

The observation scheme specifies a bandwidth budget of 4 bytes/s. The priorities of visible objects `m_state` and `election_timeout` are set to 1, the priority of `tp_leader_cycle` is set to 2, and the priority of `light` is set to 3. DDE is selected as replacement policy for all objects.

4.3 Memory Overhead

We investigate the memory (RAM, ROM) overhead of vLevels by compiling the output of the preprocessor with msp430-gcc version 3.2.3 with the `-Os` optimization switch using a June 18, 2010 CVS snapshot of Contiki.

Table 2 shows memory footprint a) without vLevels, b) with vLevels and 0 byte buffers and no visible objects, and c) with vLevels and 50 byte buffers and all four visible objects, respectively. b) results in an increase of RAM by 70 bytes and an increase of ROM by 2608 bytes. As vLevels maintains two buffers, the two 50 byte buffers in c) result in an increase of RAM by 100 bytes. Besides buffer overhead, an additional 10 bytes of RAM are required for each visible object. Additional visibility levels result in extra ROM consumption. Also, instrumentation for snapshot creation results in ROM overhead. One logging instrumentation for a 16-bit variable consumes about 40 bytes. The tracker state variable is assigned at 15 places, resulting in a ROM overhead of 600

Table 2. vLevels memory footprint for Contiki on Tmote sky

Binary	RAM (bytes)	ROM (bytes)
Tracking application	5956	29510
+vLevels 0B	6026	32118
+vLevels 50B	6166	33808

bytes. In summary, the overhead of vLevels is about 2% of the total RAM and 8% of the total ROM of a Tmote Sky for the studied application, which we find to be acceptable.

4.4 Runtime Overhead

To evaluate the runtime behavior of vLevels, we run the application with vLevels in the COOJA/MSPSim simulator (CPU speed 3.9 MHz). We only consider a single tracking node. The experiment lasts 300 seconds, the target appears at 50 seconds, remains static, and disappears at 250 seconds. We measure the vLevels overhead in terms of CPU cycles for initialization (call to `vlevel_init`), logging (invocation of `vlevel_log`), the buffer management thread (`buf_proc`), and the send thread excluding the actual transmission of the data (`logcast_proc`).

Table 3. Average cycle counts of different parts of vLevels

buffer size	vlevel_init	vlevel_log	buf_proc	logcast_proc	total	ratio
12	1716	1056	803	1803	489378	0.04%
24	1716	821	799	1775	413061	0.04%
48	1716	866	791	1723	390942	0.03%

Table 3 shows the average cycle counts of these parts of vLevels during the experiment for different buffer sizes. `vlevel_log` is invoked 130 times in total and its cycle count varies with buffer size. Small buffers result in higher overhead as the scheduler must downgrade visibility levels more frequently to fit everything into the buffer. The overheads of `buf_proc` and `logcast_proc` are independent of the buffer size, but `buf_proc` is called more often for smaller buffers. The number of invocations to `logcast_proc` is proportional to the bandwidth constraint given by users. Finally, we calculated the aggregate overhead introduced by vLevels during the complete experiment and the ratio of the aggregate overhead to the total CPU cycles as shown in Table 3, which we find to be acceptable.

4.5 Accuracy and Bandwidth

Observation is the more accurate, the higher the selected visibility levels. Figure 3 depicts the visibility levels selected by the scheduler over time for the four visible objects for different buffer sizes. The figure shows that the visibility level of `light` degrades as log entries for objects with higher priority (i.e. `m_state`, `election_timeout` are generated between time 50 and 250 seconds. Note that higher-priority objects are assigned higher visibility levels in case of larger buffer sizes as there is a bigger set of log entries to pick from when reducing visibility levels. Hence, larger buffers result in more

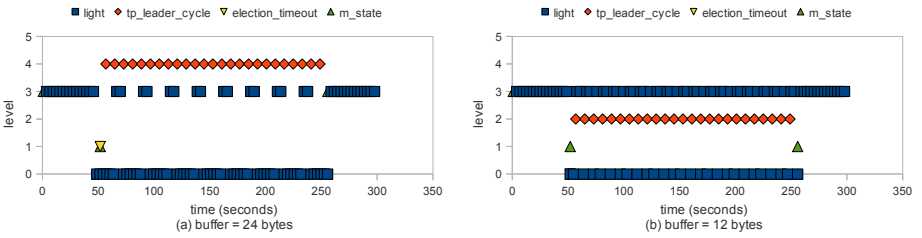

Fig. 3. Accuracy of collected data for different buffer sizes

Table 4. Throttled bandwidth and reporting latency by buffer sizes

	original data	48B buffer	24B buffer	12B buffer
bandwidth [bytes/sec]	5.53	3.5	3.24	3.94
latency [sec]	-	12	6	3

accurate observations at the cost of a longer reporting latency for the same bandwidth budget.

Table 4 shows the bandwidth of the uncompressed raw logging data and the bandwidth throttled by vLevels for different buffer sizes during the experiment. The numbers show that the throttled bandwidth is always lower than the bandwidth budget specified in the observation scheme (i.e., 4 bytes/s).

5 Related Work

In the recent past, several techniques and tools for monitoring and debugging deployed sensor networks have been proposed. Initial efforts towards debugging sensor networks are Sympathy [10] and Memento [13], both of which support the detection of a fixed set of problems. To improve repeatability of distributed event-driven applications, EnviroLog [8] provides an event recording and replay service that helps users to debug problems that are difficult to reproduce. Tools such as Marionette [17] support logging and retrieval of runtime state for fault diagnosis. Clairvoyant [18] is an interactive source-level debugger for sensor nodes. Declarative Tracepoints [2] implements a language abstraction for insertion of tracepoints into binary program images that bears some similarity to our notion of visible objects. To minimize the interference with debugged sensor networks, tools such as SNIF [11] that diagnose networks by analyzing sniffed messages can detect many failure symptoms, but visibility is limited to network messages. By allowing nodes to broadcast program state, Passive Distributed Assertions [12] support the detection of bugs caused by the incorrect interaction of multiple nodes. By optimizing log collection, LIS [14] enables users to extract detailed execution traces from resource-constrained sensor nodes. In [16], visibility is regarded as an important metric for protocol design and is improved by creating an optimal decision tree so that the energy cost of diagnosing the cause of a failure or behavior can be minimized. Unlike providing a principle for protocol design, our approach creates a mechanism to tune the visibility of internal node states. Outside the context of debugging, Energy Levels

[7] provides a programming abstraction to meet user-defined lifetime goals while maximizing application quality, which inspires the idea behind visibility levels. However, to our knowledge none of these approaches explicitly supports managing the tradeoff between visibility and resource consumption as vLevels does. Hence, we believe that vLevels is complementary to these previous techniques.

6 Conclusions

Debugging deployed sensor networks requires visibility of the node states. However, increasing visibility also incurs a higher resource consumption in terms of communication bandwidth or storage space. Especially for long-term monitoring of a sensor network it is hence crucial to find the right balance between sufficient visibility and tolerable resource consumption. Existing monitoring tools lack the ability to explicitly manage this tradeoff. We address this limitation by proposing vLevels, a 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. By means of a case study of a tracking application we showed that the memory and runtime overhead of vLevels is reasonably small and that vLevels can automatically adjust visibility to meet the resource budget.

References

1. Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Lou, L., Son, S., Stankovic, J., Stoleru, R., Wood, A.: Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In: Proc. ICDCS 2004, pp. 582–589. IEEE Computer Society, Washington (2004)
2. Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., Luo, L.: Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In: Proc. SenSys 2008, pp. 85–98. ACM, New York (2008)
3. Dunkels, A., Finne, N., Eriksson, J., Voigt, T.: Run-time dynamic linking for reprogramming wireless sensor networks. In: Proc. SenSys 2006, pp. 15–28. ACM, New York (2006)
4. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proc. LCN 2004, pp. 455–462. IEEE Computer Society, Washington (2004)
5. Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., Marrón, P.J.: Cooja/mspsim: interoperability testing for wireless sensor networks. In: Proc. SIMUTools 2009, pp. 27–27. ICST, Brussels (2009)
6. Kicales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, Springer, Heidelberg (1997)
7. Lachenmann, A., Marrón, P.J., Minder, D., Rothermel, K.: Meeting lifetime goals with energy levels. In: Proc. SenSys 2007, pp. 131–144. ACM, New York (2007)
8. Luo, L., He, T., Zhou, G., Gu, L., Abdelzaher, T.F., Stankovic, J.A.: Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In: Proc. INFOCOM 2006, pp. 1–14. IEEE Press, New York (2006)
9. Necula, G.C., Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: Parra, G. (ed.) CC 2002. LNCS, vol. 2304, pp. 209–265. Springer, Heidelberg (2002)

10. Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., Estrin, D.: Sympathy for the sensor network debugger. In: Proc. SenSys 2005, pp. 255–267. ACM, New York (2005)
11. Ringwald, M., Römer, K., Vitaletti, A.: Passive inspection of sensor networks. In: Aspnes, J., Scheideler, C., Arora, A., Madden, S. (eds.) DCOSS 2007. LNCS, vol. 4549, pp. 205–222. Springer, Heidelberg (2007)
12. Römer, K., Ma, J.: PDA: Passive distributed assertions for sensor networks. In: Proc. IPSN 2009, pp. 337–348. IEEE Computer Society, Washington (2009)
13. Rost, S., Balakrishnan, H.: Memento: A health monitoring system for wireless sensor networks. In: IEEE SECON 2006, pp. 575–584. IEEE Press, New York (2006)
14. Shea, R., Srivastava, M., Cho, Y.: Lis is more: Improved diagnostic logging in sensor networks with log instrumentation specifications. Tech. Rep. TR-UCLA-NESL-200906-01 (June 2009)
15. Tsiftes, N., Dunkels, A., He, Z., Voigt, T.: Enabling large-scale storage in sensor networks with the coffee file system. In: Proc. IPSN 2009, pp. 349–360. IEEE Computer Society, Washington (2009)
16. Wachs, M., Choi, J.I., Lee, J.W., Srinivasan, K., Chen, Z., Jain, M., Levis, P.: Visibility: a new metric for protocol design. In: Proc. SenSys 2007, pp. 73–86. ACM, New York (2007)
17. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: using rpc for interactive development and debugging of wireless embedded networks. In: Proc. IPSN 2006, pp. 416–423. ACM, New York (2006)
18. Yang, J., Soffa, M.L., Selavo, L., Whitehouse, K.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: Proc. SenSys 2007, pp. 189–203. ACM, New York (2007)