Dynamic Logging with Dylog in Networked Embedded Systems

WEI DONG, LUYAO LUO, and CHAO HUANG, Zhejiang University

Event logging is an important technique for networked embedded systems like wireless sensor networks. It can greatly help developers to understand complex system behaviors and diagnose program bugs. Existing logging facilities do not well satisfy three practical requirements: flexibility, efficiency, and high synchronization accuracy. To simultaneously satisfy these requirements, we present Dylog, a dynamic logging facility for networked embedded systems. Dylog employs several techniques. First, Dylog uses binary instrumentation for dynamically inserting or removing logging statements, enabling flexible and interactive debugging at runtime. Second, Dylog incorporates an efficient storage system and log collection protocol for recording and transferring the logging messages. Third, Dylog employs a lightweight data-driven approach for reconstructing the synchronized time of the logging messages. Dylog uses MAC-layer timestamping and drift compensation to achieve high synchronization accuracy. We implement Dylog on the TinyOS 2.1.1/TelosB platform. Results show the following: (1) Dylog incurs a small overhead. Indirections in Dylog incur an additional execution overhead of less than 1%. Dylog reduces the logging storage size by approximately 50% compared with the standard TinyOS radio printf library. Dylog reduces the patch size by more than 90%, compared with incremental reprogramming. (2) Dylog reduces the synchronization overhead by 78% in terms of transmission cost, compared with a traditional time synchronization protocol, FTSP, and it can achieve a high time synchronization accuracy of 5.4$\mu$s. (3) Dylog can help diagnose system problems effectively at the source-code level for three real-world scenarios.

Categories and Subject Descriptors: C.2.1 [Computer Communication Networks]: Network Architecture and Design—Distributed networks; D.4.7 [Operating Systems]: Organization and Design

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Networked embedded systems, diagnosis, logging

ACM Reference Format:

1. INTRODUCTION

It is often a daunting task to perform problem diagnosis in networked embedded systems like wireless sensor networks (WSNs). In these systems, the embedded devices are often inaccessible by system managers or developers once deployed (e.g., for environmental monitoring or structural protection). Getting diagnostic data from these distributed and resource-constrained devices is very difficult. While significant research in networked embedded systems has been devoted to creating intricate tools to help
diagnosis and debugging, logging remains one of the primary means for developers to get insights into the complex system behaviors [Sauter et al. 2011]. Our experience in managing two large-scale sensor network systems, GreenOrbs [Liu et al. 2011] and CitySee [Mao et al. 2012], shows that it is indeed important to inspect the system behaviors by examining important event logs. The availability of the event logs is a prerequisite for advanced diagnosis and analysis (e.g., frequent pattern mining [Khan et al. 2008], principal component analysis [Xu et al. 2009; Dong et al. 2013]) at the backend.

Unlike logging facilities for PCs, the logging facility for resource-constrained networked embedded systems needs special considerations. For example, the widely used TelosB nodes in WSNs have 10kB RAM, 48kB ROM, and 1MB external flash [Moteiv Corp. 2004]. Traditional logging is inappropriate since it consumes too much storage space [Xu et al. 2009]. In addition, these nodes typically collaborate with each other in a distributed manner, interacting with the unpredictable environment in a complex way. Performing spatial-temporal analysis on multiple logs requires accurate time synchronization among these logs.

Existing logging facilities for networked embedded systems are insufficient for real-world systems. In particular, we are facing the following practical requirements.

—First, the logging facility should be flexible, allowing for dynamically inserting or removing logging statements. This is especially important for deployed systems since it is difficult to add logging statements at the right places beforehand.

—Second, the logging facility should have an efficient way to store the logging messages and to automatically collect them from multiple nodes for later inspection.

—Third, the logging messages should be of high quality. In particular, the logging timestamps at multiple nodes should be accurately synchronized so that complex network interactions can be reasoned about.

These requirements are not well satisfied by existing approaches. DT [Cao et al. 2008] focuses on dynamic tracing, but its tracing capability is still restricted; for example, it does not allow accessing local variables or function arguments. DT does not address the second and third requirements. Envirolog [Luo et al. 2002] improves repeatability of experimental testing of sensor networks via asynchronous event recording and replay. TinyLTS [Sauter et al. 2011] enables network-wide logging for WSNs. Both Envirolog and TinyLTS lack the ability of dynamically adding or removing logging statements.

In this work, we propose a logging facility called Dylog to address all three requirements. Dylog employs several techniques to enable lightweight and interactive logging. First, Dylog uses binary instrumentation for dynamically inserting or removing logging statements, enabling interactive debugging at runtime. Second, Dylog incorporates an efficient storage system for recording the logging messages. In addition, Dylog employs a lightweight log collection protocol. Dylog significantly reduces the communication cost by storing string identifiers and restoring them back to corresponding strings at the PC. Third, Dylog employs a lightweight data-driven approach for reconstructing the synchronized timestamps of the logging messages with high precision.

While Dylog borrows ideas from existing literature, we believe it has unique contributions in achieving flexible, lightweight, and high-quality logging for networked embedded systems. The contributions of this work are summarized as follows:

—First, to our knowledge, we are the first to propose a systematic approach for addressing all three requirements.

—Second, we employ a binary instrumentation method based on text comparison with symbol address substitution for achieving dynamic logging, allowing flexible variable access and flexible insertions of logging statements.
—Third, we propose a lightweight data-driven approach for reconstructing the synchronized timestamps of logging messages. To achieve high accuracy, we employ MAC-layer timestamping and drift compensation to deal with time uncertainties in low-duty-cycle networks.

—Finally, we implement Dylog on the TinyOS 2.1.1/TelosB platform. Dylog greatly improves the system visibility for understanding system behaviors. We report three case studies of applying Dylog in the development of the GreenOrbs application [Liu et al. 2011], showing that Dylog is able to quickly localize the source-level bugs, including an unrevealed bug in the TinyOS STM25P driver.

The rest of this article is organized as follows. Section 2 introduces the related work. Section 3 describes the motivations. Section 4 presents Dylog’s design details. Section 5 shows the evaluation results. Finally, Section 6 concludes this article and gives future research directions.

2. RELATED WORK
The development of networked embedded systems like WSNs is difficult due to factors such as resource constraints and their highly distributed nature. There exist a number of ready-to-use systems or networking components. Operating systems such as TinyOS [Hill et al. 2000], Contiki OS [Dunkels et al. 2004], and SenSpire OS [Dong et al. 2011] provide basic services for programming individual devices. Networking protocols such as CTP [Gnawali et al. 2009], Drip [Tolle and Culler 2005], and Deluge [Hui and Culler 2004] provide basic services for developing distributed applications. However, the deep understanding of the software remains challenging, making effective diagnosis difficult. In the following subsections, we discuss various related works that are most pertinent to ours.

2.1. Sensor Network Diagnosis
There is a rich literature in diagnosing WSNs. Sympathy [Ramanathan et al. 2005] collects network metrics periodically and employs a decision tree for failure detection. PAD [Liu et al. 2010] uses lightweight network monitoring and Bayesian-network-based analysis to infer network failures and their causes. While these approaches can narrow down the source of a failure to a node or a communication path, they cannot localize the fault at the code level. T-check [Li and Regehr 2010] and Kleenet [Sasnauskas et al. 2010] use code analysis techniques for discovering code-level faults. While static analysis is useful, it usually suffers from the state exposition problem. It also loses the invaluable runtime information.

Clairvoyant [Yang et al. 2007] enables interactive debugging at sensor nodes. Clairvoyant incurs relatively large runtime overhead. DT [Cao et al. 2008] employs an SQL-based language interface for debugging, integrating the benefits of previous debugging techniques. The tracing capability of DT is still restricted; for example, it does not allow accessing local variables or function arguments.

In our previous work, we proposed D2 [Dong et al. 2013], an anomaly detection and diagnosis method for deployed sensor networks. D2 employs binary instrumentation to perform lightweight function count profiling. Based on the statistics, D2 uses a PCA (principal component analysis)-based approach for automatically detecting network anomalies. Compared with previous methods, D2 is able to point programmers closer to the most likely causes by a novel approach combining statistical tests and program call graph analysis. Dylog complements D2 in two ways. First, Dylog can provide useful event logs to D2, further enhancing D2’s diagnostic ability. Second, D2 can only narrow down the problem into function levels. Dylog can further disclose the potential bugs by logging inside the functions.
2.2. Tracing and Logging on Sensor Nodes

There is a growing interest in tracing and logging in networked embedded systems. Sundaram et al. [2010] use the BL algorithm [Ball and Larus 1996] to trace the intraprocedural and interprocedural control flow. Aveksha [Tancreti et al. 2011] exploits the on-chip debug module and uses an additional FPGA board to perform nonintrusive tracing on a sensor node. These approaches either cause large runtime overhead or demand special hardware.

LIS [Shea et al. 2010] is a lightweight logging tool that allows insertion of low-overhead logging calls into a system. LIS is implemented using CIL (C intermediate language) [Necula et al. 2002]. Enviroleg [Luo et al. 2002] improves repeatability of experimental testing of sensor networks via asynchronous event recording and replay. Application programmers need to specify annotations to the source code. TinyLTS [Sauter et al. 2011] enables network-wide logging for WSNs. It offers an option to store the logging messages on the external flash and employs a log collection protocol (called LTP) to collect the messages to a central sink. All of these above works lack the ability of dynamically adding or removing logging statements, and they are thus difficult to apply in a deployed system in which the logging requirement cannot precisely predetermined.

2.3. Log-Based Diagnosis

The logging messages contain useful information for discovering system problems. DustMiner [Khan et al. 2008] identifies bugs in sensor network software by checking discriminative log patterns. DustMiner requires detailed logging that relies on logging tools.

In traditional systems, programmers usually rely on logging for inspecting detailed system behaviors. Xu et al. [2009] mine console logs in data center servers to find performance anomalies. Lou et al. [2010] propose a method to mine interleaved traces to localize program bugs. LogEnhancer [Yuan et al. 2011] automatically “enhances” existing logging code to aid in future postfailure debugging. SherLog [Yuan et al. 2010] analyzes source code by leveraging information provided by runtime logs to infer what must or may have happened during the failed production run. It infers both control and data value information regarding the failed execution. While the logging facility on PCs is well developed and can greatly help in problem diagnosis, it needs special considerations for networked embedded systems like WSNs because of the severe resource constraints and the highly distributed nature.

2.4. Summary

We are aiming to propose an efficient logging facility, Dylog, for networked embedded systems. Table I compares Dylog with related approaches in some key aspects. In summary, Dylog has three important features that distinguish it from prior work: dynamic logging, efficient log collection, and lightweight log synchronization. Dynamic logging is important to interactively add or remove logging statements to a deployed system. Compared with existing works employing binary modification, Dylog uses a method based on text comparison with symbol address substitution, allowing for inserting logging statements at arbitrary code positions. Log synchronization is important to infer network-wide behaviors. Compared with Enviroleg’s log synchronization, Dylog’s method does not require global synchronization in the network. Instead, it uses a lightweight data-driven approach for synchronizing multiple logs. We believe that the logging ability offered by Dylog, combined with advanced log analysis techniques [Xu et al. 2009; Lou et al. 2010; Yuan et al. 2010], can greatly improve the software reliability of future networked embedded systems.
3. MOTIVATION
In this section, we use real-world scenarios to illustrate the need for the requirements described in Section 1.

3.1. Dynamic Logging
GreenOrbs is a large-scale sensor system consisting of more than 300 nodes deployed in the forest [Liu et al. 2011]. In the GreenOrbs network, each node employs the collection tree protocol (CTP) [Gnawali et al. 2009] for data collection. Each node chooses its parent node, which has the minimum path ETX toward the sink node. The ETX metric [Couto et al. 2003] denotes the expected number of transmissions and it is an additional metric. The ETX of a path can be obtained by summing up the ETX of all links composing the path. CTP is a dynamic routing protocol (i.e., a node can change its parent), in which the ETX metric is estimated dynamically by exploiting both the control-plane traffic and the data-plane traffic.

In the software design, we have already included a few network metrics (e.g., task duty cycle, radio duty cycle, the number of parent changes, etc.) in order to monitor the system performance. After deployment, we observe some unexpected network behaviors; for example, some nodes frequently change their parents.

In this scenario, it would be useful to dynamically add some logging statements in order to precisely record the parent change events at relevant nodes. Currently, it is nontrivial to make software changes to a deployed system. A possible solution is to use wireless reprogramming techniques, for example, Deluge [Hui and Culler 2004]. However, wireless reprogramming usually demands a hardware reboot. It is highly probable that some particular events no longer exist after reboot. Besides, it incurs a large communication overhead, making the solution undesirable for real-world applications.

3.2. Log Collection
A simple way of retrieving logs is via the serial connection. But this is only possible in indoor testbeds. For outdoor deployments, most nodes may not be easily accessible. Therefore, there is a need for efficiently storing the logging messages on local nodes.

For an operational WSN system like GreenOrbs, it is desired that the logging messages can be collected by wireless communications. This way, we can reduce the human efforts to manually collect the nodes back for log retrieval.

Table I. Comparison of Related Approaches

<table>
<thead>
<tr>
<th></th>
<th>Binary modification</th>
<th>Multihop log collect.</th>
<th>Log sync.</th>
<th>Log analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>TinyLTS</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LIS</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>EnviroLog</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Aveksha</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Dustminer</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Clairvoyant</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>D2</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dylog</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

A tick symbol indicates the availability of the corresponding feature and a cross symbol indicates the unavailability of the corresponding feature.
3.3. Time Reconstruction

In managing GreenOrbs, we observe an average network delivery of about 82% [Dong et al. 2014b]. It would be interesting to understand when and where packets are lost. The network metrics collected in GreenOrbs can reveal a proportion of loss events and their possible root causes [Dong et al. 2014b]. Still, a proportion of loss events are left unclear. In this scenario, it would be useful to obtain event logs with synchronized timestamps so that we can perform spatial-temporal correlations to investigate the interactions of multiple network elements.

Another strong motivation of synchronized time reconstruction is to understand the possible transmission collisions among neighboring nodes. A collision event can only be detected when two transmission events occur very close in time. In this scenario, we need stringent synchronization accuracy among logged events at neighboring nodes. Packet collision could also be a major cause of packet loss.

4. DESIGN

In this section, we present the design of Dylog. Section 4.1 gives an overview of Dylog. Section 4.2 describes how Dylog performs dynamic instrumentation in order to install the logging statements. Section 4.3 introduces how Dylog efficiently stores the logging messages and how we automatically collect them back to the sink node. Section 4.4 describes how Dylog accurately synchronizes multiple event logs from different nodes.

4.1. Overview

Figure 1 gives an overview of Dylog. On the PC side, DyDiff compares two versions of code and generates a patch for installing at each node. The patch is then transferred to each node via a dissemination protocol like Deluge [Hui and Culler 2004]. The DyAgent, residing on the node, installs the patch and starts executing the new code. The logging messages are efficiently stored on the external flash of the node. Once requested, the logging messages are collected back to the PC via a log collection protocol. On the PC side, the logging messages are reconstructed with the correct format and the synchronized timestamps. In the following subsections, we elaborate on each individual aspect of Dylog.
4.2. Binary Instrumentation

The process of binary instrumentation is described as follows. First, application programmers add or remove logging statements at the source level. The original and new source files are compiled into executable files. The executable files are further disassembled into assembly files. Our script, DyDiff, is used to identify the “real” differences, fetch the corresponding binary code, and prepare the final patch using a technique called trampoline [Ekman and Thane 2007]. The patch is then transferred to sensor nodes where the node-side DyAgent performs the actual patching and starts executing the new instrumented code.

Dylog incorporates several design decisions. First, Dylog obtains the binary patch by comparing the assembly files of two versions. Second, with Dylog, nodes are able to log messages in the external flash, and the messages can be acquired later by the sink. Third, Dylog preprocesses the logging statements so that string identifiers (instead of the strings, e.g., “counter=%d, counter+1=%d\n”) are stored and transferred. After retrieving the messages at the PC, the logging messages are reconstructed by mapping string identifiers to the corresponding strings. By this mechanism, Dylog significantly reduces the cost for storage and communication.

Note that Dylog’s method of obtaining the binary patch differs from DT [Cao et al. 2008]. In DT, the newly inserted code is programmed in a declarative language called TraceSQL, which is then compiled by the TraceSQL compiler. Our benefit is twofold. First, we reduce the engineering efforts of writing a new customized compiler. Second, our approach can precisely locate the binary insertion point. In contrast, it would be difficult for DT to precisely locate the binary insertion point if the code is inserted at locations other than the beginning or end of a function. This is because the TraceSQL compiler compiles the inserted code without its context in the entire program.

We use an example to show the process of binary instrumentation. The original code is a simple TinyOS application that increments a counter every second. Now we would like to add one logging statement to the original code. The following code snippet shows how we add the logging statement at the source level.

```c
event void MilliTimer.fired() {
    counter++;
    logNow("counter=%d, counter+1=%d\n",
            counter, counter+1);
}
```

DyDiff uses the following steps to generate the final patch.

4.2.1. Preprocessing. First, Dylog preprocesses the logging statement by replacing the string constant to a string identifier. Therefore, the original logNow(...) shown previously is replaced by log-Now(ID, counter, counter+1), where ID denotes the string identifier. We use the order with which the string appears in the source code as the string identifier. The string identifier can uniquely identify the string constant in the source file.

4.2.2. Disassembling. Second, we disassemble both versions of code. The disassembled result from msp430-objdump for the original file is:

```
Original code:
4828: 11 20 jnz $+36
482a: 92 53 36 11 inc $0x136 ; counter++
482e: 02 3c jmp $+6
4830: 5b 53 inc.b r11
```
Note that this code is inlined in the function `VirtualizeTimerC__0__fireTimers`. The disassembled result for the new code is:

<table>
<thead>
<tr>
<th>New code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4828: 11 20 jnz $+36</td>
</tr>
<tr>
<td>482a: 92 53 36 11 inc &amp;0x1136 ; counter++</td>
</tr>
<tr>
<td>&gt;&gt;482e: 1f 42 36 11 mov &amp;0x1136,r15 ; r15=counter</td>
</tr>
<tr>
<td>&gt;&gt;4832: 1f 53 inc r15</td>
</tr>
<tr>
<td>&gt;&gt;4834: 0f 12 push r15</td>
</tr>
<tr>
<td>&gt;&gt;4836: 12 12 36 11 push &amp;0x1136 ; counter</td>
</tr>
<tr>
<td>&gt;&gt;483a: 30 12 a8 47 push #0</td>
</tr>
<tr>
<td>&gt;&gt;483e: b0 12 22 5b call #23330 ; &lt;logNow&gt;</td>
</tr>
<tr>
<td>&gt;&gt;4842: 31 50 06 00 add #6, r1</td>
</tr>
<tr>
<td>4846: 02 3c jmp $+6</td>
</tr>
<tr>
<td>4848: 5b 53 inc.b r11</td>
</tr>
</tbody>
</table>

We can see the changes in the assembly code. The value of counter is assigned to register r15. Then, the value of r15 increments by 1. Three parameters of logNow are pushed onto the stack from the right to the left. Note that #0 is the string identifier for the string—counter=\%d, counter++=\%d\n. call #23330 invokes the logNow function. Finally, three parameters are popped from the stack (r1 is the stack pointer register).

4.2.3. Finding “Real” Differences. An implementation challenge is to automatically find such differences. Although a direct text comparison on the assembly instructions can find these differences, it also incurs a huge amount of false positives. This is because it will also find a lot of unnecessary differences caused by code shift. For example, a call instruction to a function located after the insertion point (0x482e) will be identified. This is unnecessary because we will not “reprogram” the entire new code; instead, we would like to perform small modifications at the insertion point while preserving the basic program layout.

In order to eliminate such false positives, we substitute relative addresses to symbols before performing text comparison on the assembly code.

After address substitution, the text comparison can find out the “real” differences:

<table>
<thead>
<tr>
<th>Real difference:</th>
</tr>
</thead>
<tbody>
<tr>
<td>482e: 1f 42 36 11 mov &amp;0x1136,r15 ; r15=counter</td>
</tr>
<tr>
<td>4832: 1f 53 inc r15</td>
</tr>
<tr>
<td>4834: 0f 12 push r15</td>
</tr>
<tr>
<td>4836: 12 12 36 11 push &amp;0x1136 ; counter</td>
</tr>
<tr>
<td>483a: 30 12 a8 47 push #0</td>
</tr>
<tr>
<td>483e: b0 12 22 5b call #23330 ; &lt;logNow&gt;</td>
</tr>
<tr>
<td>4842: 31 50 06 00 add #6, r1</td>
</tr>
</tbody>
</table>

4.2.4. Generating the Patch. Before we explain the format of the patch, it is necessary to understand how Dylog instruments the original binary code in order to execute the newly added logging statements. We use a technique called trampoline [Ekman and Thane 2007]. The basic principle is to direct the control to the added code, which is allocated at the end of the program, and finally, to return back to the insertion point to continue the original execution.

It is worth noting some key benefits of our employed technique [Ekman and Thane 2007] compared with other existing techniques, for example, ‘C [Engler et al. 1996] and DynInst [Buck and Hollingsworth 2000]. First, our technique allows adding logging statements at user-preferred insertion points at runtime without preparation of the original source code. ‘C [Engler et al. 1996] requires decisions to be made prior
Fig. 2. Binary instrumentation using trampoline. There is a call instruction at the insertion point (line 482e); lines 621a–622e denote the “real differences” by comparing the original and new assembly files; lines 6232–6236 denote the “mirrored” instructions of lines 482e–4830 in the original assembly file; finally, there is a ret instruction to return the control to the instructions following the call instruction.

There are several implementation details worth noting. First, if we have to replace relative instructions, we cannot simply mirror the instructions at a different location. Instead, we should “translate” the instructions to use the absolute address. In the example shown in Figure 2, the relative jmp instruction `jmp $+6` is translated to an absolute branch instruction `br 0x4834`. Second, we should preserve the “inline” property of the original code so that the differences can be kept small. Otherwise, if the code is originally inlined and happens to be non-inlined in the newly compiled code, there would be a large amount of unnecessary differences.

Now we return to the explanation of patch format. It is clear from Figure 2 that there are two segments for one added logging statement. The first segment encodes the call instruction and the second segment encodes the “real” difference, the “mirrored” instructions, and the ret instruction. Generally, for \(N\) added logging statements, there would be \(N + 1\) segments. The first \(N\) segments correspond to \(N\) call instructions at the insertion points. The last (large) segment corresponds to code allocated at the end of the program, performing the real logging actions (e.g., lines 621a–6238).

The patch format is encoded in binary using a format called .raw. The .raw file (i.e., patch) consists of multiple sections. Each section is described with a section length,
memory location, and the real binary code. A single patch usually consists of two sections. For the example shown in Figure 2, the patch is as follows:

| 04 00 2E 48 B0 12 1A 62 |
| 20 00 1A 62 20 00 1F 42 36 11 1F 53 0F 12 12 12 |
| 36 11 30 12 A8 47 B0 12 22 5B 31 50 06 00 30 40 |
| 34 48 5B 53 30 41 00 00 00 00 00 00 00 00 |

There are two sections. The first section corresponds to line 482e in the new assembly code (see Figure 2). Its length is 4 bytes, and the start memory location is 0x482E. The second section corresponds to lines 621a–6236 in the new assembly code (see Figure 2). Its length is 0x20 bytes, and the start memory location is 0x621A.

The patch can be transferred to each node. The DyAgent residing on each node performs the actual patching. DyAgent maintains patching information. In particular, it additionally maintains the patch that can undo the actions. For example, before overwriting memory according to patch p, DyAgent first saves the original memory contents in another patch, p’. This way, patch p can be removed later by overwriting the memory according to patch p’. Moreover, the system can always roll back to the original state.

4.3. Efficient Logging and Collection

We provide several logging mechanisms for common use.

1. Serial connection. The logging messages are directly transferred to PC via serial connections. This is suitable for predeployment testing in which nodes are easily accessible.

2. External flash. The logging messages are stored on the external flash of nodes. The logging messages can later be retrieved by either serial connection or wireless communication. The first method requires collecting all nodes back.

3. External flash+wireless communication. The logging messages are stored on the external flash and collected to a central sink via wireless communications.

We focus on the third method since it is the most convenient way of retrieving logging messages. We illustrate our design using the TinyOS operating system. Note that our design principle can be applied elsewhere.

4.3.1. Log Storage. TinyOS provides three abstractions for accessing the external flash. The ConfigStorageC component provides services for storing configuration data. The LogStorageC component provides methods to record fixed-size logs. The BlockStorageC component provides random access to a block of storage. In our case, variable-sized messages need to be stored. For this purpose, we use the more general BlockStorageC component. Based on this component, we implement an additional component for accessing variable-sized records.

The format of a specific record is depicted in Figure 3. The length field consumes 1 byte and is used to indicate the length of the record. The localtime field consumes 4 bytes and is used to indicate the timestamp of the log. The ID field consumes 2 bytes
and is used to uniquely identify the string constant. The variable field is for storing the variable number of parameters in the logging message.

For a typical record involving two variables of the \texttt{uint16_t} type, the record consumes 11 bytes. For the TelosB nodes with 1MB external flash, a total of approximately 95,000 records can be stored. Once the maximum number of records is reached, old records will be overwritten by new ones. We also use a RAM buffer to store the latest logging messages in order to handle a burst of log events.

It is worth emphasizing two optimizations to reduce the storage cost in our implementation. First, we do not explicitly store the string constants. Instead, we store string identifiers on each node. Second, we support variable-length records, which can use the external flash more efficiently.

4.3.2. Log Collection. We design a particular log collection protocol based on CTP [Gnawali et al. 2009] to collect the logging messages back. It provides two message retrieval strategies: active and passive.

In the \textit{active} approach, each node periodically sends logging messages to the central sink. Similar to Sauter et al. [2011], it provides priority-based packet transmissions. The logging packets are enqueued to the sending queue if and only if there is no application messages enqueued.

In the \textit{passive} approach, a specified node or a specified subset of nodes sends a bulk of the logging messages to the central sink after the sink issues a log retrieval command. We can improve the efficiency of log retrieval by disabling low-power listening typically employed in WSNs. The network can be tuned back to the low-power mode once the log retrieval is finished.

We use the retransmission strategy to achieve a high log collection reliability. If critical logging messages are lost on the way, the sink node can issue a command via the existing dissemination protocol for requesting these messages. We note that other protocols can also be employed, for example, Flush [Kim et al. 2007], which can provide reliable bulk data transfer using techniques like end-to-end acknowledgments.

4.4. Time Reconstruction

Dylog adopts a lightweight data-driven approach for reconstructing the synchronized time for logging messages (with respect to the sink’s time).

4.4.1. MAC-Layer Timestamping. Dylog exploits existing data traffic in a collection network. It appends a 4-byte timestamp to each data packet originated at each source node to keep synchronization between the source node and the sink node. In this way, when the logging messages are retrieved once requested, the local timestamps in these messages can be automatically synchronized to the sink node.

MAC-layer timestamping relies on the underlying CC2420 SFD (start frame delimiter) interrupt mechanism: during packet transmission, when a preamble of a packet is transmitted, an SFD (start frame delimiter) interrupt is generated immediately; during packet reception, when the preamble of a packet is received, an SFD interrupt is also generated immediately. Since the signal propagation time over short distance (as is the case for the 802.15.4-compliant CC2420 chip) is negligible, SFD at the sender and SFD at the receiver essentially correspond to the same event. This forms the basis for accurate time synchronization.

Suppose we have a packet \texttt{pkt}_A from A with a three-hop routing path—(A-B-C). The node sojourn time $d_X(pkt_A)$ (where $X$ denotes the node ID) can be measured with MAC-layer timestamping.

—At the source node A, $d_A(pkt_A)$ measures the time duration from the instant when the network layer sends the packet to the instant when the packet preamble is transmitted over the radio chip.
—At the forwarding node B, \( d_B(pkt_A) \) measures the time duration from the instant when the radio chip receives the packet preamble to the instant when the corresponding packet preamble is transmitted over the radio chip.

—At the sink node C, \( d_C(pkt_A) \) measures the time duration from the instant when the radio chip receives the packet preamble to the instant when the network layer receives the packet.

An estimation of the network sojourn time of a packet (i.e., packet delay) is simply the sum of all sojourn times along the routing path. The measured packet delay using MAC-layer timestamping captures the nondeterministic delays at local nodes including software routines and MAC backoffs. Suppose increased network traffic leads to network congestion and collisions. The forwarding node B retransmits the packet until it is successfully delivered to the sink node C. Let’s use \( t_B^x(pkt_A) \) to denote the time instant when the packet is received at node B, \( t_B^{x1}(pkt_A) \) to denote the time instant when the packet is transmitted over the radio chip for the first time, and \( t_B^{x2}(pkt_A) \) to denote the time instant when the packet is transmitted over the radio chip for the second time. The first transmitted packet (failed) includes node sojourn time \( d_B^1(pkt_A) = t_B^{x1}(pkt_A) - t_B^{x2}(pkt_A) \), while the second transmitted packet (successful) includes node sojourn time \( d_B^2(pkt_A) = t_B^{x2}(pkt_A) - t_B^2(pkt_A) \). We can see that \( d_B^1(pkt_A) \) in the second successfully transmitted packet already includes the time due to MAC retransmissions since \( t_B^{x2}(pkt_A) \) records the time instant when the radio chip actually transmits the packet after potential backoffs.

We exploit existing data packets to piggyback the information for lightweight time synchronization at the sink. If no data packets are transmitted to the sink, we can generate additional control packets to carry the needed information. The time reconstruction accuracy of Dylog can be affected by lost data packets since the number of anchor time pairs is reduced when packets are lost. However, practical sensor networks usually employ a retransmission strategy to achieve a relatively high data collection reliability. We can still achieve a high reconstruction accuracy.

### 4.4.2. Drift Compensation.

A simple approach to perform synchronization at the sink node is to attach in each packet a local generation time field, denoting the time when the network layer sends the packet at the source node. For example, we denote it as \( t_g(pkt_A) \) in our three-node example. At the sink side, we know the packet reception time \( t_s(pkt_A) \). Since we know the packet delay \( d(pkt_A) \), we can estimate the packet generation time with respect to sink as \( t_s(pkt_A) - d(pkt_A) \). So we get an anchor time pair \((t_g(pkt_A), t_s(pkt_A) - d(pkt_A))\) (where C denotes the sink node) each time the sink receives a data packet from a source node.

This simple approach has problems in accurately estimating packet delays since the node sojourn time is counted using each node’s local clock. It may suffer from clock drift especially for low-duty-cycle sensor networks with long forwarding delays.

We exploit data traffic in existing networks to compensate for clock drift. In Figure 4, B estimates its clock skew with respect to A, exploiting the two latest packet receptions. A informs B about the interval between the transmissions of the two packets in the MAC layer, denoted as \( \tau_A \). B measures the interval between the receptions of the two packets in the MAC layer, denoted as \( \tau_B \). Assuming 802.15.4 links, we consider that both the transmit SFD and receive SFD interrupts happen at the same time. The clock skew of B with respect to A is thus estimated as \( \alpha_A(B) = \tau_B/\tau_A \). The accumulated delay at B is no longer \( d_{AB} = d_A + d_B \) (we omit \( pkt_A \) to simplify the notation). Rather, it is the estimation of accumulated delay with respect to B, that is, \( d_{AB} = d(A) \cdot \alpha_A(B) + d(B) \). Node C further estimates the delay as \( d_{ABC} = d_{AB} \cdot \alpha_B(C) + d_C \), where \( \alpha_B(C) \) is C’s clock skew with respect to B using packets from B to C. We can see that Dylog uses clock skew estimation to get an accurate estimate of packet delay.
4.4.3. Log Synchronization. With the previous approach, the sink can regularly obtain anchor time pairs from data packets from a given source node. With these anchor time pairs, \((E^1_A, E^1_C), (E^2_A, E^2_C), (E^3_A, E^3_C), \ldots\) (here we use A to denote the source node and C the sink node, \(E^i_X\) denotes the packet generation time with respect to node X for the \(i\)th packet from A), we can use the linear clock model to compensate for the effect of clock drift. In particular, in order to recover the synchronized timestamp of a logging message, we find the anchor time pairs closest to the logging message within a time window. Then we build a linear clock model to convert the local timestamps in the logging messages to the synchronized timestamps with respect to the sink node.

Suppose the event time of a log is \(L_A\). We find anchor time pairs \((E^i_A, E^i_C)(1 \leq i \leq n)\), which satisfy \(|L_A - E^i_A| \leq W\), where \(W\) denotes the time window size. We would like to estimate \(L_C\), that is, the log event time with respect to node C.

We first build a linear clock model \(t_C = \alpha t_A + \beta\), where \(\alpha\) denotes the clock skew and \(\beta\) denotes the clock offset. This clock model is estimated using the anchor time pairs.

In order to minimize the estimation error, \(\alpha\) and \(\beta\) need to satisfy the following conditions. First, the expectation of \(t_C\) equals that of \(E^i_C\):

\[
\frac{1}{n} \sum_{i=1}^{n} (\alpha E^i_A + \beta) = \frac{1}{n} \sum_{i=1}^{n} E^i_C. \tag{1}
\]

Second, \(\alpha\) and \(\beta\) minimize the deviation of \(t_C\) to \(E^i_C\):

\[
(\alpha, \beta) = \arg\min_{\alpha, \beta} \sum_{i=1}^{n} ((\alpha E^i_A + \beta) - E^i_C)^2. \tag{2}
\]

Solving Equations (1) and (2) yields the value of \(\alpha\) and \(\beta\) as follows:

\[
\alpha = \frac{\sum_{i=1}^{n}(E^i_A - \overline{E}_A)(E^i_C - \overline{E}_C)}{\sum_{i=1}^{n}(E^i_A - \overline{E}_A)^2}, \tag{3}
\]

\[
\beta = \overline{E}_C - \alpha \overline{E}_A.
\]

where \(\overline{E}_A\) and \(\overline{E}_C\) are mean values of \(\{E^i_A\}_{i=1}^{n}\) and \(\{E^i_C\}_{i=1}^{n}\).

Once the clock model is estimated, we can use it to calculate \(L_C\), which equals \(\alpha L_A + \beta\).

5. EVALUATION

In this section, we evaluate Dylog from three aspects. Section 5.1 examines Dylog’s overhead. Section 5.2 conducts a comparative study to show the benefits of Dylog compared with existing approaches like incremental reprogramming. Section 5.3 presents three case studies to show Dylog’s ability in localizing system problems.
We perform the evaluation on the TinyOS/TelosB platform with an 8MHz msp430x1611 microprocessor (with 10kB RAM, 48kB ROM, and 1MB external flash) and a 250kbps CC2420 radio chip [Moteiv Corp. 2004].

5.1. Overhead
In this section, we examine Dylog’s execution overhead, I/O overhead, and log collection overhead, as well as the overhead of time reconstruction.

5.1.1. Execution Overhead. In order to evaluate Dylog’s execution overhead, we add one logging statement in a function. We evaluate the execution overhead of:

— the original function without insertion,
— the modified function with recompilation, and
— the modified function with Dylog.

Table II shows the results. We can see two overheads involved in Dylog: (1) the overhead of invoking the logging function that imposes an overhead of $2.773 - 2.750 = 0.23$ ms, which is unavoidable if we need logging, and (2) the overhead of indirection in Dylog that introduces an additional cost of $2.775 - 2.773 = 0.002$ ms (i.e., $2\mu s$)—this overhead is very small.

5.1.2. I/O Overhead. We measure the overhead of I/O operations on the external flash. Table III shows the results. Writing 1 byte incurs an average delay of 0.059ms, while reading 1 byte incurs an average delay of 0.056ms. Considering a logging message with 11 bytes, the writing delay is approximately 0.6ms. During this interval, the current of the node increases to 12mA. The I/O cost of logging is much larger than the CPU cost on TelosB nodes.

5.1.3. Collection Overhead. We consider the overhead of collecting the logging messages by wireless communication. The overhead depends on both the number of messages and the performance of the collection protocol. Assume each node logs 1,000 records, with each consuming 11 bytes.

We evaluate the throughput of the log collection protocol in a testbed consisting of 24 TelosB nodes. The nodes employ the CTP protocol [Gnawali et al. 2009] for data collection. Figure 5 shows how the receiving throughput varies with increasing sending rate at each node.

The maximum throughput is approximately 38kbps. This means that our protocol would spend $11 \times 1000 \times (24 - 1) \text{bytes} / (38/8) \times 1000 \text{bytes/second} \approx 53$ s to collect all messages back. Normally, not all logging messages are useful for inspection. If a
relatively small amount of logging messages need to be collected from a subset of nodes, this overhead is still acceptable.

5.1.4. Time Reconstruction Overhead. We conduct experiments in a 24-node testbed to compare Dylog with FTSP [Maróti et al. 2004]—a traditional time synchronization protocol for accurate time reconstruction. We note that FTSP was employed by EnviroLog [Luo et al. 2002] for accurate log synchronization.

In the experiments, we use a modified TestNetwork program in TinyOS. One TestNetwork application employs Dylog’s synchronization method, while another employs the FTSP protocol. The transmission power is set to $-31.5$ dBm in order to simulate multihop behaviors. Each node periodically sends three data packets to the sink node via multihop wireless (using the CTP protocol [Gnawali et al. 2009]) with a data transmission period of 10 seconds. Note that many real WSN systems collect data in a periodic fashion with a uniform distribution in time. For example, [Chipara et al. 2010] present a clinical monitoring system with continuous and periodic sensing and collection. Barrenetxea el al. [2008] present SensorScope, in which sensor nodes periodically forward gathered data to the sink for environmental monitoring. To monitor CO$_2$ absorption in the forest, we deployed the GreenOrbs network [Liu et al. 2011] with a data collection period of 10 minutes. It is worth noting that our approach is not restricted to periodic sensor networks.

We use the default FTSP protocol in TinyOS. Figures 6(a) and 6(b) compare the overhead of Dylog with FTSP in terms of the number of transmissions and the radio duty cycle for an experimental duration of 2 hours. We can see that Dylog’s method has a much smaller overhead than FTSP since it does not generate additional traffic for time synchronization. For example, we can see from Figure 6 that Dylog reduces the transmission cost by 78% compared with FTSP. Therefore, the passive approach adopted by Dylog is suitable for deployed sensor networks for its low overhead.

We also evaluate the synchronization accuracy of the MAC-layer timestamping mechanism with a Tektronix TDS3034C oscilloscope. We measure the time difference of two synchronized events at both the sender and the receiver. Results show that the sender and receiver can achieve a synchronization accuracy of approximately $5.4\mu$s. The maximum clock drift is 40ppm as specified in the Telos datasheet [Moteiv Corp. 2004]. Consider a scenario in which a packet takes 1 hour to travel from the source to the sink node. Without drift compensation, the maximum synchronization error could reach $3600 \times 40/1000000 = 0.144$ second in the worst case. With drift compensation, the
synchronization errors do not accumulate significantly even with a relatively long forwarding delay. We also evaluate the log synchronization accuracy in our 24-node testbed using the same TestNetwork program. With the transmission power of $-31.5\text{dBm}$, we find that our method can achieve an accuracy below $50\mu\text{s}$.

5.2. Comparative Study

We conduct a comparative study from two aspects. First, we evaluate the benefits of Dylog compared with the standard TinyOS printf library. Second, we evaluate the benefits of Dylog compared with incremental reprogramming.

5.2.1. Dylog Versus TinyOS Radio Printf. Table IV shows the average logging message length in TinyOS major components including CTP [Gnawali et al. 2009], DIP [Lin and Levis 2008], Deluge [Hui and Culler 2004], and DHV [Dang et al. 2009]. With normal logging using TinyOS radio printf, the average length is relatively large. Dylog reduces approximately 50% overhead by eliminating string constants. This is important since it reduces both the storage cost and the communication cost.

5.2.2. Dylog Versus Incremental Reprogramming. We compare Dylog with incremental reprogramming since incremental reprogramming can also modify the software after deployment. We consider three change cases in the CTP component:

—Small change case: four logging statements are added.
—Median change case: 26 logging statements are added
—Large change case: 40 logging statements are added.

Table IV. Comparison with Normal Logging Layer

<table>
<thead>
<tr>
<th>Component</th>
<th>printf (bytes)</th>
<th>Dylog (bytes)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drip [Tolle and Culler 2005]</td>
<td>49.00</td>
<td>19.00</td>
<td>2.58</td>
</tr>
<tr>
<td>Deluge [Hui and Culler 2004]</td>
<td>21.09</td>
<td>12.80</td>
<td>1.65</td>
</tr>
<tr>
<td>DHV [Dang et al. 2009]</td>
<td>32.49</td>
<td>12.88</td>
<td>2.52</td>
</tr>
</tbody>
</table>
Table V. Number of Bytes Required to Be Transferred

<table>
<thead>
<tr>
<th>Case</th>
<th>Reprogramming</th>
<th>Dylog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small change</td>
<td>3,032</td>
<td>122</td>
</tr>
<tr>
<td>Median change</td>
<td>14,294</td>
<td>928</td>
</tr>
<tr>
<td>Large change</td>
<td>15,603</td>
<td>1,406</td>
</tr>
</tbody>
</table>

Table V shows the number of bytes required to be transferred. We can see that Dylog reduces the overhead by more than 90% compared with the RMTD incremental reprogramming approach [Hu et al. 2009]. This is because Dylog preserves the memory layout of the original program so that the modifications can be greatly reduced.

To see the impact on the dissemination process, we evaluate the dissemination time and the number of transmitted data packets to “reprogram” the nodes with traditional incremental reprogramming and Dylog. We use Deluge [Hui and Culler 2004] as the code dissemination protocol.

Figures 7(a) and 7(b) show the dissemination time and the number of transmissions in a single-hop network with two TelosB nodes. Figures 7(c) and 7(d) show the dissemination time and the number of transmissions in a testbed consisting of 24 TelosB nodes, with a transmission power of $-31.5$ dBm. We can see that Dylog significantly reduces the overhead to “reprogram” the nodes.

In order to see the benefits of Dylog compared with incremental reprogramming in a general setting, we propose a model to analyze the overhead for disseminating the patch. Figure 8 shows the new code layout after employing Dylog and incremental
reprogramming. The white part indicates the unchanged part, while the gray part indicates the changed part. We assume the original code is $C$ bytes, and we need to install $N$ logging statements. With Dylog, the transferred overhead is $O_{\text{dylog}} = LN$, where $L$ is the overhead for a single patch. For the example shown in Figure 2, $L = 44$ bytes.

With incremental reprogramming, the analysis is more complicated. We further assume the number of reference instructions is $M$. Note that reference instructions will change due to code shift, causing additional overhead. With incremental reprogramming, we usually use the COPY command to copy continuous bytes from the original code for reconstructing the new code. In addition, we use the ADD command to insert bytes that do not appear in the original code [Hu et al. 2009; Mo et al. 2014]. Without loss of generality, we assume the COPY command consumes 5 bytes, while the ADD command consumes the number of bytes that need to be inserted [Hu et al. 2009].

The $N$ logging statements separate the code into $N + 1$ segments in the new code. Ideally, the $N + 1$ segments can be copied from the original code, while the $N$ logging statements need to be added. The complexity is that the insertion of logging statements causes code shifts that change the target fields in the reference instructions. A key question is, how many reference instructions are changed? It is worth noting that instructions referencing the first segment do not change since the first segment does not shift.

We assume that each segment is equally likely referenced. The number of changed reference instructions equals the number of instructions referencing the remaining $N$ shifted segments. Thus, the number of changed reference instructions is $MN/(N+1)$. In the worst case, the code is separated into $N + \frac{MN}{N+1}$ segments. In the best case, the code is separated into $\max\{N, \frac{MN}{N+1}\}$ segments. The address field is 2 bytes for TelosB. The overhead is thus $O_{\text{inc}}^w = 5 \cdot (N + \frac{MN}{N+1}) + LN + 2 \cdot \frac{MN}{N+1}$ in the worst case, and $O_{\text{inc}}^b = 5 \cdot \max\{N, \frac{MN}{N+1}\} + LN + 2 \cdot \frac{MN}{N+1}$ in the best case, where $L' = 24$ bytes denotes the single patch overhead for the example shown in Figure 2 using the COPY and ADD encoding method. Note that $L' < L$ since Dylog’s patch (in .raw format) encodes additional memory addresses.

In order to investigate how Dylog’s approach decreases the transferred overhead for practical applications, we first examine the number of reference instructions (e.g., call, br for TelosB nodes) for typical benchmarks. Figure 9 shows the results. We can see that the number of reference instructions occupy approximately one-third of the total number of instructions. Figure 10(a) shows the transferred overhead with
5.3. Case Studies

In this section, we present three case studies to show Dylog’s ability in diagnosing code bugs during the development of a real-world WSN system, GreenOrbs [Liu et al. 2011].

5.3.1. Trickle Timer in TinyOS 2.1.0. We see the following symptom in our application, which already includes a number of metrics for inspecting overall system performance. When we burn a set of nodes in the testbed, we find that the task duty cycle increases to nearly 88% after 10 minutes. The task duty cycle is measured by counting the time ticks spent in executing TinyOS tasks. Normally, the node executes the program code with a low task duty cycle of approximately 3%.

Another benefit of Dylog is that unlike incremental reprogramming, it does not require node reboot. This is important in two aspects. First, the cost of rebooting can be avoided [Dong et al. 2010]. Second, and more importantly, the program state will be preserved. Otherwise, the logging of particular events might be impossible after reboot.
In order to see which task is frequently executed, we add the following logging statement in the TinyOS scheduler:

```c
command bool TaskScheduler.runNextTask() {
    uint8_t nextTask;
    atomic {
        nextTask = popTask();
        if (nextTask == NO_TASK) {
            return FALSE;
        }
    }
    signal TaskBasic.runTask[nextTask]();
    >>logNow("Task %d executed\n", nextTask);
    return TRUE;
}
```

After inspecting the logging messages, we find that task ID 33 (TrickleTimerImplP$0__timerTask__runTask) is frequently running after 10 minutes. Therefore, we identify that the bug is in the Trickler timer component. However, without detailed information, we still cannot localize the bug.

After statically parsing the source code (app.c), we find that the function TrickleTimerImplP$0__timerTask__runTask posts this task. Thus, we guess that the Trickle timer firing interval might be incorrectly generated, causing the fired events frequently running.

Hence, we further add the following logging statement in the function generateTime(), which generates the firing interval:

```c
trickles[id].remainder =
    (trickles[id].period << scale) - newTime;
trickles[id].time += newTime;
>>logNow("period=%d, scale=%d, \n    newTime=%d, remainder=%d\n",
    >>    trickles[id].period, scale,
    >>    newTime, remainder);
```

We find that when period = 64 (and scale = 10), newTime = 64,640 (0xFC80) and remainder = 4,294,902,656 (0xFFFF0380). The value of remainder happens to be the complementary code of newTime; that is, the (trickles[id].period << scale) becomes zero. We notice that trickles[id].period is of type uint16_t, so if it is 64 and it is shifted to the left by 10, the 1s are pushed out.

This bug is fixed in TinyOS 2.1.1 by modifying the statement to the following code:

```c
trickles[id].remainder =
    (((uint32_t)trickles[id].period)<<scale) - newTime;
```

This case shows that real bugs are typically incrementally narrowed down. Dylog allows remote modification of logging statements, which is especially important for deployed sensor networks. This is the main advantage of Dylog compared with static logging facilities such as TinyLTS [Sauter et al. 2011] and EnviroLog [Luo et al. 2002]. With Dylog, we can conveniently and incrementally dispatch the logging statements until the bug is identified. The dispatching overhead is small. Without Dylog, we have to resort to reprogramming a debug version each time we would like to change something, incurring a much larger overhead. More importantly, some problems may not exist once the node reboots.
5.3.2. Broken External Flash. When developing Dylog itself, we encounter the following symptom. A subset of nodes have a task duty cycle of approximately 88%. These nodes cannot transfer any logging messages.

An initial guess is that there exist some tasks repeatedly running on nodes with broken external flash. However, without detailed logging information, we cannot localize the bug in the source code.

With Dylog, we can drill down this problem. Since we cannot log messages on the external flash this time, we transfer the logging messages directly via the serial connection.

First, we need to identify which task is frequently running. We add a logging statement in the TinyOS scheduler as described previously.

After static code analysis, we instrument functions satisfying two requirements: (1) related to the external flash and (2) involved in the call graph of the frequently running task. We find that the function Stm25pSpiP__SpiResource__granted is repeatedly executed. Inside this function, releaseAndRequest is repeatedly invoked.

```
    event void SpiResource.granted() {
        if ( !_m_is_writing )
            signal ClientResource.granted();
        else if ( sendCmd( 0x5, 2 ) & 0x1 )
            releaseAndRequest();
        else
            signalDone( SUCCESS );
    }
```

At this time, we could guess the causes of the bug: when the code powers up the external flash, it does not check the status of the hardware. Therefore, if the external flash is broken and _m_is_writing=True, the code would repeatedly make requests to acquire the resource.

We indeed find the bug in the Spi.powerUp function.

```
    async command error_t Spi.powerUp() {
        uint8_t signature;
        signature = sendCmd( S_POWER_ON, 5 );
        return SUCCESS;
    }
```

The function always returns SUCCESS without checking the value of the signature. We check the STM25P data sheet [STMicroelectronics 2004] and find that the powering up succeeds only when the signature equals 0x13.

Based on this finding, we make revisions to STM25P code. Basically, we should record whether it is successful to power up the flash. If it fails, we shall not make further requests. The revised powerUp function is as follows:

```
    async command error_t Spi.powerUp() {
        uint8_t signature;
        signature = sendCmd( S_POWER_ON, 5 );
        if ( signature == 0x13 ) {
            return SUCCESS;
        }
        else {
            m_cancel = TRUE;
            return FAIL;
        }
    }
```
The revised `SpiResource.granted` function is as follows:

```c
event void SpiResource.granted() {
    if ( !m_is_writing )
        signal ClientResource.granted();
>>else if (m_cancel)
>>    return;
    else if ( sendCmd( 0x5, 2 ) & 0x1 )
        releaseAndRequest();
    else
        signalDone( SUCCESS );
}
```

This bug still exists in the latest TinyOS code, and we plan to make notifications to the TinyOS development group.

5.3.3. Link Estimation in TinyOS 2.1.1. We encounter the following symptom when we deploy a sensor network consisting of 50 nodes in the forest. After the system is deployed, some nodes experience heavy packet losses. We add some nodes for relaying. But the nodes with poor performance seem very reluctant to select the relay nodes.

We guess that there might be some inefficiencies in the link estimation code, which again affects the route selection.

We add logging statements in the `4bitle` component to record the link qualities.

```c
if (totalPkt == 0)
    ne->inquality = (ALPHA * ne->inquality) / 10;
else {
    newEst = (250UL * ne->rcvcnt) / totalPkt;
    ne->inquality = (ALPHA * ne->inquality +
            (10-ALPHA) * newEst)/10;
>>    logNow("inquality=%d, newEst=%d\n",
>>        ne->inquality, newEst);
}            
ne->rcvcnt = 0;
ne->failcnt = 0;
updateETX(ne, computeETX(ne->inquality));
```

After collecting the logging messages back, we find that the relay node's link quality (ne->inquality) is much lower than expected. The logging messages reveal that when the relay node first enters the neighbor table of the current node, its initial link quality (newEst) is combined with the initialized link quality (i.e., zero) by a moving average with ALPHA=9. This makes the average inquality very small.

The latest code fixes this bug. If it is the first time a neighboring node enters the neighbor table (i.e., the MATURE_ENTRY bit is clear), the average link quality should be initialized to the instant link quality.

```c
if (!(ne->flags & MATURE_ENTRY)) {
    newEst = (250UL * ne->rcvcnt) / totalPkt;
    ne->inquality = newEst;
    ne->etx = computeETX(ne->inquality);
}
ne->flags |= MATURE_ENTRY;
newEst = (250UL * ne->rcvcnt) / totalPkt;
ne->inquality = (ALPHA * ne->inquality +
            (10-ALPHA) * newEst)/10;
ne->rcvcnt = 0;
ne->failcnt = 0;
updateETX(ne, computeETX(ne->inquality));
```
This case emphasizes another benefit of Dylog: it automates the process of collecting and synchronizing multiple logs so that developers can focus on a consistent log for error detection.

6. CONCLUSION

In this article, we present Dylog, a dynamic logging facility for networked embedded systems. We present a systematic approach for simultaneously satisfying three requirements: flexibility, efficiency, and high synchronization accuracy. First, Dylog uses binary instrumentation for dynamically inserting or removing logging statements, enabling flexible and interactive debugging at runtime. Second, Dylog incorporates an efficient storage system and log collection protocol for recording and transferring the logging messages. In particular, Dylog significantly reduces the communication cost by storing string identifiers and restoring them back to corresponding strings at the PC. Third, Dylog employs a lightweight data-driven approach for reconstructing the synchronized time of the logging messages. Dylog uses MAC-layer timestamping and drift compensation to achieve high synchronization accuracy.

We implement Dylog on the TinyOS 2.1.1/TelosB platform. Results show the following: (1) Dylog incurs a small execution overhead. Indirections in Dylog incur an additional execution overhead of less than 1%. Dylog reduces the logging storage size by approximately 50% compared with the standard TinyOS radio printf library. Dylog reduces the patch size by more than 90%, compared with incremental reprogramming. (2) Dylog reduces the synchronization overhead by 78% in terms of transmission cost, compared with a traditional time synchronization protocol, FTSP, and it can achieve a high time synchronization accuracy of 5.4μs. (3) Dylog can help quickly diagnose system problems at the source-code level for three real-world scenarios.

As future work, it is important to extend Dylog’s instrumentation capability. We would like to support general patching to a deployed system. It is interesting to apply Dylog in real-world systems to gain more insight about network behaviors. Although Dylog is currently implemented on TinyOS, its key principles can also be applied elsewhere. We would like to extend Dylog to other OSs and platforms. Another future work is to develop advanced event mining techniques to automatically detect program anomalies.

REFERENCES


Jian-Guang Lou, Qiang Fu, Fengqi Yang, and Jiang Li. 2010. Mining program workflow from interleaved logs. In *Proceedings of ACM SIGKDD*.


Dynamic Logging with Dylog in Networked Embedded Systems


Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In Proceedings of ACM SOSP.


Received April 2014; revised April 2015; accepted July 2015