Dynamic Logging with Dylog for Networked Embedded Systems

Wei Dong†, Chao Huang†, Jiliang Wang‡, Chun Chen†, and Jiajun Bu†
†College of Computer Science, Zhejiang University
‡School of Software and TNLIST, Tsinghua University
Email: {dongw.huange}@emnets.org, jiliang@greenorbs.com, {chenc,bjj}@zju.edu.cn

Abstract—We present Dylog, a dynamic logging facility for networked embedded systems. 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 the 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 MAC layer timestamping and a linear clock model for reconstructing the synchronized time of the logging messages with a very high precision. We implement and evaluate Dylog on TinyOS 2.1.1/TelosB. Results show that Dylog incurs a reasonable runtime overhead. Dylog can help gain great visibility into the system behaviors, and diagnose performance issues at the source code level.

I. INTRODUCTION

Logging and tracing are important methods to gain insight into the behaviors of networked embedded systems like wireless sensor networks (WSNs).

Usually, tracing focuses on program control flow and is used to find out the performance bottlenecks. It is not uncommon to observe a performance degradation by a factor of 5–100 if tracing is enabled in each function or each basic block. Therefore, it is costly to perform tracing.

In the field of WSNs, a few approaches have been proposed for efficient tracing. Sundaram et al. [1] use the BL algorithm [2] to trace the intra-procedural and inter-procedural control flow. The runtime overhead is relatively large and may influence the execution timing of real applications. Tancreti et al. [3] exploit the on-chip debug module and use an additional FPGA board to perform non-intrusive tracing on a sensor node. It, however, demands special hardware which imposes limitations on the tracing expressiveness. Cao et al. [4] presents declarative tracepoints (DT) for inserting checkpoints to a deployed sensor system at runtime. The ability of adding and removing tracepoints at runtime is important as users can thus dynamically control the placement of “interested tracepoints” without redeploying and rebooting the new application. DT’s tracing capability, however, is still restricted, e.g., it does not allow accessing local variables or arguments of functions.

On the other hand, logging provides a more flexible way to record user-interested information. For example, the traditional “printf” function allows users to record interested events with relevant variables.

In the field of WSNs, there are several existing works on logging. TinyOS provides a printf library to show messages on the screen over the USB port. The radio printf library in TinyOS-contrib allows transferring messages over the radio. TinyLTS [5] 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.

Our experience in managing two large-scale sensor network systems, GreenOrbs [6] and CitySee [7], shows that it is indeed important to inspect the system behaviors by examining important event logs, while the current logging (or tracing) mechanisms are insufficient for real-world systems. First, we need a mechanism to dynamically add or remove logging statements for observing the system behaviors after deployment since it is difficult to add logging statements at the right places beforehand. Second, we need an efficient mechanism to store the messages and then to collect the messages to the central sink. Finally, as we log messages distributed across multiple nodes, we need to synchronize logs among multiple nodes to investigate the complex interactions.

The above requirements are not fully addressed in the current literature. While DT [4] allows dynamic tracing (the first requirement), its tracing capabilities are limited as mentioned above. Moreover, DT does not address the second and third requirements. TinyLTS [5] exposes an expressive interface for logging and allows efficient collection of messages (the second requirement). It, however, does not address the first and third requirements.

In this work, we aim to propose an efficient logging facility addressing all three requirements. The resulting system, called Dylog, employs several techniques to achieve the goals. First, Dylog uses binary instrumentation for dynamically inserting or removing logging statements, enabling interactive debugging at the runtime. Second, Dylog incorporates an efficient storage system and log collection protocol for recording and transferring the 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 MAC layer timestamping and a linear clock model for reconstructing the synchronized time of the logging messages with a very high precision.

We implement Dylog on TinyOS 2.1.1/TelosB. Dylog greatly improves the system visibility for understanding system behaviors and diagnosing performance issues. We report case studies of applying Dylog during the development of GreenOrbs application, showing that Dylog is able to quickly localize the bugs in the source code.

The rest of this paper is organized as follows. Section II introduces the related work. Section III describes the motivations. Section IV presents Dylog’s design details. Section V shows the evaluation results. Finally, Section VI concludes this paper and gives future research directions.

II. RELATED WORK

The development of networked embedded systems like WSNs is intrinsically difficult because of multiple impacting factors, such as resource constraints, unpredictable environment, complex interactions among network elements, etc. There exist a number of ready-to-use systems or networking components. Operating systems such as TinyOS [8], Contiki OS [9] and SenSpire OS [10] provide basic services for programming on motes. Networking protocols such as CTP [11], Drip [12], and Deluge [13] provide basic services for developing distributed applications. Yet, the understanding and performance diagnosis of the software remain challenging.

A. Sensor network diagnosis

There is a rich literature in diagnosing WSNs. Sympathy [14] collects network metrics periodically and employs a decision tree for failure detection. PAD [15] 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 [16] and Kleenet [17] use code analysis techniques for discovering code level faults. While static analysis is useful, it usually suffers from the state exposition problem and lose the invaluable runtime information.

Clairvoyant [18] enables interactive debugging at sensor motes. DT [4] integrates benefits of previous debugging techniques and proposes a SQL-based language interface for debugging. Clairvoyant incurs relatively large runtime overhead while DT has limitations for exposing arbitrary variables at arbitrary points.

B. Tracing and logging in sensor nodes

There is a growing interest in tracing and logging in networked embedded systems. Sundaram et al. [1] use the BL algorithm [2] to trace the intra-procedural and inter-procedural control flow. Tancreti et al. [3] exploit the on-chip debug module and use an additional FPGA board to perform non-intrusive tracing on a sensor node. These approaches either cause large runtime overhead or demand special hardware. TinyLTS [5] 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. It, however, lacks the ability of dynamically adding or removing logging statements, and thus is difficult to apply in a deployed networking system.

C. Log-based diagnosis

In traditional systems, programmers usually rely on logging for inspecting detailed system behaviors. The logging messages contain useful information for discovering performance issues. Xu et al. [19] mine logs in data center servers to find performance anomalies. Lou et al. [20] propose a method to mine interleaved traces to localize program bugs. LogEnhancer [21] automatically “enhances” existing logging code to aid in future post-failure debugging. SherLog [22] analyzes source code by leveraging information provided by run-time logs to infer what must or may have happened during the failed production run. It infers both control and data value information regarding to the failed execution.

While the logging facility on PCs are well developed and can greatly help performance diagnosis, it needs further efforts in WSNs because of the distributed nature and severe resource constraints.

D. Summary

Dylog is a logging facility for networked embedded systems. It has three important features that distinguish itself from prior work, i.e., dynamic logging, efficient collection, and synchronized timestamping. Dynamic logging and efficient collection are important to interactively adding or removing logging statements to a deployed system. Synchronized timestamping is important to infer network-wide behaviors. We believe that the logging ability offered by Dylog, combined with advanced log processing techniques [19], [20], [22], [23], can greatly improve software reliability of future networked embedded systems.

III. MOTIVATION

In this section, we study real-world scenarios to illustrate the need for the requirements described in Section I.

A. Dynamic logging

GreenOrbs is a large-scale sensor system consisting of more than 300 nodes deployed in the forest [6]. In the software design, we have already included a few network metrics in order to manage the system, including task duty cycle, radio duty cycle, the number of parent changes, etc.

After deployment, we observe some unexpected network behaviors, e.g., 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 non-trivial to make software changes to a deployed system. A
possible solution is to use wireless reprogramming techniques, e.g., Deluge [13]. 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, which makes the technique itself unreliable for real-world applications.

B. Log collection

A simple way of retrieving logs is via the serial connection. But it is only possible in indoor testbed. In outdoor deployment, 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 the logging messages be collected via the wireless. This way, we reduce the human efforts to collect nodes back and do not impact the way the system operates.

C. Time reconstruction

In managing GreenOrbs, we observe an average network delivery of 82%. 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. 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 collisions among neighboring nodes. A collision event can only be detected when two transmission events occur very close in time. Packet collision could also be a major cause of packet loss.

IV. DESIGN

A. Overview

Figure 1 gives an overview of Dylog. At the PC side, DyDiff compares two versions of code, and generates a patch for installing at each mote.

The patch is then transferred to each mote via a dissemination protocol like Deluge [13]. The DyAgent, residing on the mote, installs the patch and starts executing the new code. The logging messages are efficiently stored on the mote’s external flash. Once requested, the logging messages are collected back to the PC via a collection protocol like CTP [11].

At the PC side, the logging messages are reconstructed with the correct strings and synchronized timestamps.

In the following subsections, we will elaborate on each individual aspect of Dylog. Section IV-B describes how we perform binary instrumentation for applying the patch. Section IV-C describes how we efficiently store the logging messages and how we collect the messages back to the central sink. Section IV-D describes how we accurately reconstruct the synchronized timestamps.

B. 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 instrumented binary code using a technique called trampoline [24]. The binary code 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 to the sink. Third, Dylog preprocesses the logging statements so that string identifiers (instead of the strings, e.g., “counter=%d, counter+1=%d”) 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 [4]. 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 two-fold. 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 will use an example to show the process of binary instrumentation. The original code is a simple TinyOS

![Fig. 1: Dylog overview](image-url)
application which 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);
}
```

First, we preprocess the logging statement by replacing the string constant to a string identifier. Therefore, the original logNow(...) shown above is replaced by logNow(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.

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 &0x1136 ; 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:

```
New code:
4828: 1f 42 36 11 mov &0x1136,r15 ; r15=counter
4832: 1f 53 inc r15
4834: 0f 12 push r15
4836: 1f 53 36 11 inc &0x1136 ; counter
483a: 30 12 a8 47 push #0
483e: b0 12 22 5b call #23330 ; <logNow>
4842: 31 50 06 00 add #6, r1
```

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+1=%d\n. call #23330 invokes the logNow function. Finally, three parameters are popped from the stack (r1 is the stack pointer register).

We need a tool 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:

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

We use a technique called trampoline [24] to patch the difference. The basic principle is to direct the control to the added code which is allocated at the end of the program, and finally, return back to the insertion point to continue the original execution.

Figure 2 shows the trampoline technique. First, the instructions after the insertion point are replaced by a call instruction which directs the execution to the added code which is located at the end of the program. The patched code first executes the “real” difference, and then executes the replaced instructions. Finally, the ret instruction returns the execution to continue the original execution.

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. For example, in the above 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 could 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.

We use a simple binary file format called .raw to encode the binary code. The .raw file consists of several sections. Each section is described with a section length, memory location, and the real binary code. A single patch usually consists of two sections. In our example, the .raw file should be:

```
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 58 31 50 06 00 30 40
34 48 5B 53 30 41 00 00 00 00 00 00 00
```

There are two sections. The length of the first section is 4 bytes, and the start memory location is 0x482E. The length of the second section is 0x20 bytes, and the start memory location is 0x621A. The first section is used to replace two instructions in the original code in order to direct the execution to the patch. The second section is used to add the binary code at the end of the program.

The .raw file can be transferred to each mote. The DyAgent residing on each node performs the actual patching. DyAgent also maintains patching information. In particular, it additionally maintains the patch that will undo the actions. This way, the corresponding patch can later be removed. Moreover, the system can always rollback to the original state.

### C. Efficient logging and collection

We provide several logging implementations for common use.

1) Serial connection. The logging messages are directly transferred to PC via serial connections. This is suitable for pre-deployment testing in which motes are easily accessible.

2) External flash. The logging messages are stored on the external flash of motes. 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 while at the same time incurs large overhead without optimization.

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-sized 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 one byte and is used to indicate the length of the record. The localtime field consumes four bytes and is used to indicate the timestamp of the log. The ID field consumes two 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 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 mote. Second, we support variable length records, which can use the external flash more efficiently.

We design a particular log collection protocol based on CTP [11] to collect the logging messages back. It provides two message retrieval strategies, i.e., active and passive.

In the active approach, each node periodically sends logging messages to the central sink. Similar to [5], it provides priority-based packet transmissions. The logging packets are only enqueued to the sending queue if and only if there is no application messages enqueued.

In the passive approach, a specified node or a specified subset of nodes send a bulk of logging messages to the central sink after the sink issues a log retrieval command. We improve the efficiency of log retrieval by disabling low power listening. The network can quickly tuned back to low power mode once the communication finishes and the request is satisfied.

There is another issue worth mentioning here. Each time, we would like to install the patch, we should first uninstall all previous patches. This is because Dylog compares assembly files of the original version (version 0) and the current version (version current). Dylog does not know the exact memory layout of the previous version (version current-1).

This may cause a performance issue. Assume patches NO1 and NO2 are installed on the nodes. Now we would like to install patches NO2, NO3, NO4 to the nodes. A direct approach is to send all three patches to the network. A more efficient way is to send just two required patches, i.e., NO3 and NO4 to the network. Dylog adopt the latter approach. For this purpose, DyAgent maintains the latest received patches so that they can be installed later on.

Dylog only sends the minimal required patches to the network, incurring small communication cost. After receiving the required patches, DyAgent first uninstalls all previous
patches, and then installs the newly received patches (NO2 and NO4 in this example) as well as the requested patches already on the motes (NO2 in this example).

D. Time reconstruction

Dylog adopts an accurate and efficient way for time reconstruction. It only adds four bytes timestamp in a subset of packets. Once the logging messages are collected at the sink, the local timestamps are automatically synchronized to the sink node.

We use the TinyOS TimeSyncPacket interface to perform MAC-level timestamping. At a high level, we modify the collection protocol to provide a send command with the localtime of an event E (e.g., node A’s local time) as a parameter. Later, when the receiver, say B, receives the packet, it can infer the timestamp of E in B’s local time.

Figure 4 shows the underlying mechanism. Node A first records the event time as $E_A$. It then invokes the send command with $E_A$ as a parameter. When the packet is transmitted by the radio, the SFD interrupt time ($SFD_A$) is recorded. The packet is extended with four additional bytes with the value of $(E_A - SFD_A)$.

When the packet is received at node B, the SFD interrupt time is recorded ($SFD_B$). Note that the signal propagation speed is very fast. Hence we consider both SFD interrupts happen at the same time. Therefore, the event time at node B can be inferred as

$$SFD_B + (E_A - SFD_A)$$  

We revise the collection protocol further to make the time reconstruction mechanism work in the multihop scenario.

Figure 5 shows how to reconstruct the event time in a multihop scenario. Nodes A, B, C form a forwarding path. First, node A transmits a packet with $E_A - SFD_A$ embedded in the packet. Second, node B infers the event time $E_B$ according to Eq. 1. Node B forwards the packet with $E_B - SFD_B$. Finally, when node C receives the packet, it infers the event time as

$$E_C = SFD_C + (E_B - SFD_B)$$  

Another important issue is that when the difference of $SFD_A$ and $E_A$ is very large, the reconstructed timestamp may be inaccurate since the clock drift cannot be neglected. On the other hand, it is quite possible since we will very likely retrieve the logging messages very infrequently.

To address this issue, each node with Dylog sends timestamped packets periodically with small difference of $SFD_A - E_A$. The timestamps can be piggybacked to application packets if possible. The reconstructed $E_B$ can thus be considered accurate.

With these anchor time pairs, $(E^i_A, E^i_B)$, $(E^2_A, E^2_B)$, $(E^3_A, E^3_B)$, ..., we can use the linear clock model to compensate 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 estimate the corresponding timestamp synchronized to the sink.

Suppose the event time of a log is $L_A$. We find anchor time pairs $(E^i_A, E^i_B)$ ($1 \leq i \leq n$) which satisfies $|L_A - E^i_A| \leq W$ where W denotes the time window size. We would like to estimate $L_B$, i.e., the log event time at the sink node B.

We first build a linear clock model $t_B = \alpha t_A + \beta$ where $\alpha$ denotes the clock drift 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 $t_B$ equals that of $E^i_B$.

$$1 \sum_{i=1}^n (\alpha E^i_A + \beta) = 1 \sum_{i=1}^n E^i_B$$  

Second, $\alpha$ and $\beta$ minimize the deviation of $t_B$ to $E^i_B$.

$$(\alpha, \beta) = \arg \min \sum_{i=1}^n ((\alpha E^i_A + \beta) - E^i_B)$$  

Solving Eqs. 3 and 4 yields the value of $\alpha$ and $\beta$ as follows,

$$\alpha = \frac{\sum_{i=1}^n (E^i_A - \bar{E}_A) (E^i_B - \bar{E}_B)}{\sum_{i=1}^n (E^i_A - \bar{E}_A)^2}$$  

$$\beta = \bar{E}_B - \alpha \bar{E}_A$$

where $\bar{E}_A$ and $\bar{E}_B$ are mean values of $(E^i_A)_{i=1}^n$ and $(E^i_B)_{i=1}^n$.

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

**Fig. 4: MAC-level timestamping**

**Fig. 5: The use of MAC-level timestamping in Dylog with multihop forwarding**
Table 1: Execution overhead (ms)

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Recompiled</th>
<th>Dylog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.750</td>
<td></td>
<td>2.773</td>
<td>2.775</td>
</tr>
</tbody>
</table>

Table 2: I/O overhead

<table>
<thead>
<tr>
<th>Notation</th>
<th>value</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I&lt;sub&gt;read&lt;/sub&gt;</td>
<td>5mA</td>
<td>current when reading bytes from external flash</td>
</tr>
<tr>
<td>I&lt;sub&gt;wrt&lt;/sub&gt;</td>
<td>12mA</td>
<td>current when writing bytes to external flash</td>
</tr>
<tr>
<td>t&lt;sub&gt;read&lt;/sub&gt;</td>
<td>0.056ms</td>
<td>average time to read one byte from external flash</td>
</tr>
<tr>
<td>t&lt;sub&gt;wrt&lt;/sub&gt;</td>
<td>0.059ms</td>
<td>average time to write one byte from external flash</td>
</tr>
</tbody>
</table>

V. Evaluation

In this section, we evaluate Dylog from three aspects. Section V-A examines Dylog’s overhead as well as the accuracy of time reconstruction. Section V-B conducts comparative study to show benefits of Dylog compared to TinyOS radio printf and incremental reprogramming. Section V-C presents case studies to show Dylog’s ability in localizing performance bugs.

A. Overhead

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.
- The modified function with Dylog.

Table 1 shows the results. We can see two overheads involved in Dylog. (a) The overhead of invoking the logging function which imposes an overhead of 2.773-2.750=0.23ms. This is unavoidable if we need logging. (b) The overhead of indirection in Dylog introduces an additional cost of 2.775-2.773=0.002ms=2µs. This overhead is very small.

2) I/O overhead: We measure the overhead of I/O operations on the external flash. Table 2 shows the results. Writing one byte causes an average delay of 0.059ms while reading one byte causes 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 mote increases to 12mA. We can see that the I/O cost of logging is much larger than the CPU cost.

3) Accuracy of time reconstruction: We also evaluate the synchronization accuracy of the MAC-level 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 5.36µs (see Figure 6).

B. Comparative study

We conduct comparative study from two aspects. First, we evaluate the benefits of eliminating string constants. Second, we evaluate the benefits of Dylog compared to incremental reprogramming.

1) Dylog vs. TinyOS radio printf: Table 3 shows the average logging message length in TinyOS major components including CTP [11], DIP [25], Drip [12], Deluge [13], and DHV [26]. With normal logging like TinyOS radio printf, the average length is relatively large. Dylog reduces approximately 50% overhead. This technique is important since it reduces both the storage cost and the communication cost.

2) Dylog vs. incremental reprogramming: We next 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: 4 logging statements are added.
- Median change case: 26 logging statements are added.
- Large change case: 40 logging statements are added.

Table 4 shows the number of bytes required to be transferred. We can see that Dylog significantly reduces the overhead compared to the RMTD incremental reprogramming approach [27]. This is because Dylog preserves the memory layout of the original program so that the modifications can be greatly reduced.

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 [28]. Second, more importantly, program state will be preserved. Otherwise, the logging of particular events might...
be impossible after reboot.

C. Case Studies

In this section, we present case studies to show Dylog’s ability in diagnosing performance bugs during the development of a real-world WSN system, GreenOrbs [6].

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 mote executes program with a low task duty cycle of approximately 3%.

In order to see which task is frequently executed, we add the following logging statement to 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_impl$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 following function—TrickleTimerImplP_impl$0___Timer__fired—post this task. Thus, we guess the Trickle timer firing interval might be incorrectly generated, causing the fired events repeatedly running.

Hence, we 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 ,
newTime= %d , remainder= %d\n",
trickles[id].period, scale,
newTime, remainder);
```

2) Link estimation in TinyOS 2.1.1: We encounter the following symptom when we have deployed a sensor network in the forest. After the system is deployed, some nodes experience heavy packet loss. We add some nodes for relaying. But the nodes with poor performance seem very reluctant to select the relay nodes.

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, we find that the relay node’s link quality (newEst) is much lower than expected. The logging messages reveal that when the relay node first enters the neighbor table of the current node, its instant 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 issue. 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;
    ne->rcvcnt = 0;
    ne->failcnt = 0;
    updateETX(ne, computeETX(ne->inquality));
}
```

We find that when period=64 (and scale=10), newTime=64640 (0xFC80) and remainder=4294902656 (0xFFFF0380). The value of remainder happens to be the complementary code of newTime, i.e., the (trickles[id].period << scale) becomes zero. trickles[id].period is of type uint16_t, so if it is 64 and 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;
```

Table 4: The 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>
VI. CONCLUSION

In this paper, we present Dylog, a dynamic logging mechanism for networked embedded systems. 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 the 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 MAC layer timestamping and a linear clock model for reconstructing synchronized timestamps of the logging messages with a very high precision. We implement Dylog on TinyOS 2.1.1/TelosB. Evaluation results show that Dylog incurs a reasonable overhead. Dylog can help to gain great visibility into the system behaviors, and diagnose performance issues at the source level.

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 insights of network behaviors. Another future work is to develop advanced event mining techniques to detect program anomalies.

ACKNOWLEDGEMENT

This work is supported by the National Science Foundation of China Grants No. 61202402, 61202359, and 61373166, the Fundamental Research Funds for the Central Universities, the Research Fund for the Doctoral Program of Higher Education of China (2012011120179), and the Demonstration of Digital Medical Service and Technology in Destined Region.

REFERENCES