LibReplay: Deterministic Replay for Bug Hunting in Sensor Networks

Olaf Landsiedel, Elad Michael Schiller, Salvatore Tomaselli

Chalmers University of Technology, Sweden
{olafl, elad}@chalmers.se and tiposchi@tiscali.it

Abstract Bug hunting in sensor networks is challenging: Bugs are often prompted by a particular, complex concatenation of events. Moreover, dynamic interactions between nodes and with the environment make it time-consuming to track and reproduce a bug. We introduce LibReplay to ease bug hunting in sensor networks: it provides (1) lightweight and flexible logging and (2) deterministic replay. LibReplay logs function calls to and from the application or another code of interest. It enables deterministic replay of execution traces in a controlled environment such as a full-system simulator. This allows the user to benefit from well-established debugging tools such as stepping through code, breakpoints, or watchpoints. We show that the lightweight architecture of LibReplay provides the benefits of replay debugging at an efficiency that is comparable to traditional logging tools, which commonly do not allow replay debugging.

Keywords: Cyber Physical Systems, Internet of Things, Wireless Sensor Networks, Debugging, Replay, Tracing, Logging, Simulation

1 Introduction

Bug hunting in sensor networks is challenging: (1) sensor networks are distributed and deeply embedded into a non-deterministic environment. (2) The non-determinism of both the wireless network and the physical environment makes it time-consuming to track and reproduce bugs. (3) Bugs are often prompted by a particular, complex concatenation of events. Source-level debugging capabilities as common in sequential programming, i.e., local and non-distributed applications, would significantly ease the debugging process. For example, stepping through code, breakpoints, and watchpoints are well-established tools to debug sequential code. However, the distributed and embedded nature prevents us from pausing program execution on a node to examine its state.

Large-scale distributed systems on the Internet solve this issue by employing logging and replay capabilities [5,8,9]. These log all interaction between the code of interest and the system itself, e.g., function calls to and from a part of an application that is suspected to malfunction. Next, they replay the execution of the code of interest accordingly to the logged function calls and their parameters. As a result, the local replay can be debugged using well-established debugging tools such as GDB and allows for stepping through code, breakpoints,

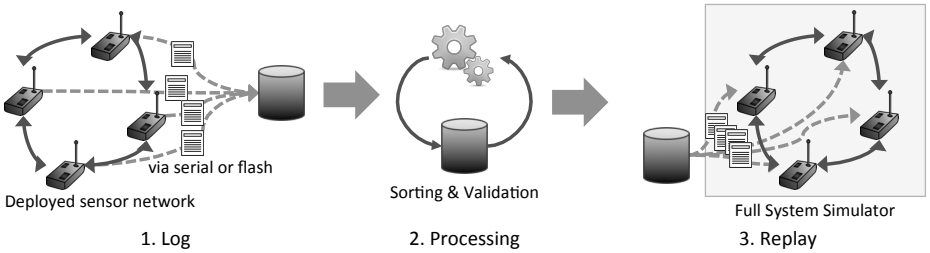


Figure 1. LibReplay in a nutshell: (1) Distributed logging via serial or flash, (2) sorting and validation of logs, and (3) replay in a full-system simulator.

and watchpoints. While this technique is well-known in large-scale distributed systems, we see limited application in the area of sensor networks due to the resource limitations of sensor nodes.

This paper closes this gap and provides LibReplay, a lightweight and deterministic solutions for distributed logging and source-level replay (see Figure 1). It allows debugging of sensor network applications and protocols with source-level debuggers, such as GDB.We achieve this by replaying execution traces in a full-system simulator, such as Cooja [10], MspSim [4], Avrora [16], or QEMU [1]. This paper makes three contributions:

- **Lightweight, Distributed Logging:** we introduce a system architecture for distributed, lightweight logging that is customizable to code regions of interest. It employs a two-phase logging to reflect resource constraints and to minimize the side effects of logging on program execution.
- **Deterministic Replaying:** From the logs we replay all input events to the code of interest. Using full-system simulators we enable deterministic, high fidelity replay. Utilizing the debugging capabilities of these platforms, one can now step through source-code and employ breakpoints and watchpoints.
- **Implementation and Performance Evaluation:** We demonstrate a working implementation of LibReplay with an efficiency that is on par with traditional logging tools, which commonly do not provide replay capabilities. We evaluate LibReplay's performance with respect to MCU and memory before showing that its overhead is similar to today's logging approaches, which cannot provide the same functionality.

Next, we discuss the limitations of traditional debugging tools and outline the differences of LibReplay to the state of the art (Section 2). We then introduce LibReplay in detail (Section 3) and compare the performance of LibReplay with the state of the art (Section 4) before addressing future directions and concluding (Section 5).

2 Limitations of the State of the Art

Logging and tracing are two common approaches for hunting bugs in sensor networks. Logging tools [2,6,11] record execution details. Commonly, they store the

log in the flash memory for off-line collection or feed them to the host system via the serial port. In practice, bug hunting with such logging tools often follows an iterative approach: (a) adding or refining logging statements, (b) re-executing the system until the bug is triggered, and (c) analyzing the log and spotting bug appearances. The developers have to repeat these steps until they understand the bug causes, try to fix them and then check whether all bugs were removed by again repeating these steps. Moreover, the non-deterministic and dynamic nature of the wireless network and interactions with the environment make it time consuming to reproduce a bug sufficiently often for this repetitive approach. In contrast, LibReplay logs function calls and their parameters to and from the code of interest, such as a malfunctioning routing protocol. As a result, LibReplay collects sufficient information to replay program execution deterministically allowing one to employ source-level debuggers for bug hunting. In our experience, this limits the need for repeated testing, and in most cases a single logging run is sufficient to fix the bug in replay debugging, because analysis and bug spotting is mainly carried out off-line using an iterative debugger that replays the log.

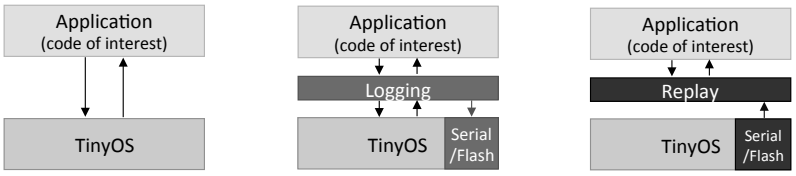
Tracing tools [12,14,15,17,18] follow a different approach: They trace the program execution by logging function calls. For example, a tracer logs each function and its parameters that a packet takes on its path through the protocol stack from the application to the radio driver. A key challenge is that tracing program execution leads to large traces when compared to traditional logging [13]. Some approaches [12,15] address this challenge with additional hardware on the nodes. For example, Minerva [12] connects a dedicated debugging-board to the JTAG adapter of the sensor node. Controlling multiple debugging-boards over Ethernet, Minerva can examine network-wide state. LibReplay, in contrast, merely logs function calls and their parameters to and from the code of interest, limiting its intrusiveness while not requiring additional hardware.

3 LibReplay: Design and System Architecture

We start the discussion of LibReplay by illustrating its basic idea before introducing LibReplay in detail.

3.1 Basic Idea: Flexible Logging and Deterministic Replay

With LibReplay, we log function calls to and from a user-specified code-region of interest, such as a malfunctioning routing protocol. In the replay, we feed the calls back to the code of interest in the same order as they were logged on the real system (see Figure 2). Thus, in the replay every event happens in the same order as on the real system and with the same function parameters. Using cycle-accurate simulation of the entire system, each event will also take the same number of cycles as on the real system. Thus, a complete log that includes all functions to the code of interest generates a complete replay with all local states equaling to the ones of the real-system. We note that due to



(a) Unmodified. (b) Logging. (c) Replay.

Figure 2. We log function calls to and from the code of interest, such as a malfunctioning routing protocol. For replay, we feed the logs back to the code of interest. Replay in a full-system simulator, such as Cooja, provides us with well-established debugging tools such as stepping through code, breakpoints and watchpoints.

the run-to-completion semantics, e.g., tasks in TinyOS, of many OSs for Cyber-Physical Systems (CPS) and Internet of Things (IoT) we do not have to log the OS scheduler itself.

3.2 Lightweight and Flexible Logging

The first building block of LibReplay is its lightweight, flexible logging-architecture. It has three design goals: (1) to reduce the overhead of logging to limit potential side-effects on program execution, (2) to provide distributed logging of events across multiple nodes, and (3) to ease integration into user-defined applications and components.

Deferred logging to limit side-effects on applications: Whenever a function to or from the code-region of interest is called, LibReplay logs the function, its parameters, the return value, and a logical timestamp, i.e., an event sequence-number. To limit run-time overhead, LibReplay employs a two-phase approach to logging: As a first step, any log data is merely buffered in RAM and the execution can continue with only minimal delay. As second step, a deferred, background process – only scheduled if no other process is to be scheduled – handles the storage itself: it moves the log buffers to flash or the serial port for storage.

Distributed logging of events across multiple nodes: When testing and debugging distributed systems, we experienced it as essential that we can trace events and messages across multiple nodes. For example, we often needed to trace how a single message travels through the network and which state changes it triggered along its path, such as timeouts and re-transmissions. To trace events across multiple nodes, LibReplay adds a logical timestamp to each outgoing radio message, which is send by the code of interest. This allows to create a globally valid order of the events for replay. Optionally, LibReplay can also re-use sequence numbers and source addresses that most protocols already provide to identify messages and their order uniquely. This avoids overhead, as no additional timestamps need to be added to messages.

Easy to integrate into user-defined applications and components: When designing LibReplay, we put a special focus on its ease and flexibility of use. For

Listing 1.1. Without logging

```
[...]

App.Receive -> AMReceiverC;

[...]
```

Listing 1.2. With logging

```
[...]
components new ReceiveLogC() as Log;
App.Receive -> Log;
Log.Receive -> AMReceiverC;
[...]
```

Table 1. LibReplay logging example: Without (left) and with (right) logging of the `Receive` interface. Adding logging to applications requires merely few changes to the wiring of TinyOS applications. Common logging components, such as the `ReceiveLogC` component used in this example to log the `Receive` interface, are provided by LibReplay.

example, LibReplay can be easily integrated into own applications and tailored by adding own logging interfaces. LibReplay places a logging component between each interface of the code of interest and the OS, see Table 1. LibReplay provides logging components for common interfaces of TinyOS.

3.3 Processing the Logs: Globally-Ordered Replay

Once events are collected from the individual nodes, we utilize the logical timestamps to construct a globally-ordered replay. Events such as radio events have (or can have) a counterpart on the other nodes, such as a transmit and receive event. This guides LibReplay to obtain a global order of events [7].

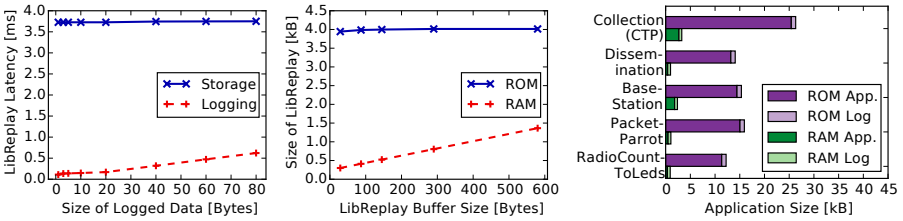
3.4 Deterministic Replay

For replay, we replace each logging component with its counterpart replay-component. Similar to the logging components, we have one replay component per interface and LibReplay provides these for the common interfaces in TinyOS. Thus, we replay the code of interest and feed it the events we previously logged.

Compared to the logging components, the data flow is now reversed and replay components feed events to the application (see Figure 2c). Bug hunting can now utilize the advanced debugging capabilities of modern system simulators such as monitoring of individual variables and stepping through code fragments. Note that when performing such tasks on the deployed systems directly, they cause high overhead and significant side effects. Additionally, we use the recorded output to detect deviations between the log and the replay, which can indicate subtle system bugs such as buffer overflows, etc. Note that the main replay-target of LibReplay are full-system simulators, as these can replay multiple nodes, and we can analyze their interaction. However, LibReplay can also replay the execution on a real node and we can connect and debug via JTAG, for example.

3.5 Discussion: Generic Design

In TinyOS, modules are the natural integration points for logging. They encapsulate local state, and state changes are only triggered via their interfaces.



(a) MCU load when a logging a function call: Logging is completed within 1ms. The low-priority background task handles the heavy lifting.

(b) The RAM footprint of LibReplay mainly depends on the size of the logging buffers. ROM remains constant independent of buffer size.

(c) The overall memory footprint of LibReplay is small when compared to the application itself (default setting, 300 bytes buffer).

Figure 3. MCU and memory overhead of LibReplay.

Nonetheless, the design of LibReplay is generic and is not bound to TinyOS. For example, instead of interfaces we can log traditional function calls to and from a block of code. This, for example, matches the design of other common OS in CPS and IoT such as Contiki [3] or FreeRTOS.

4 Evaluation

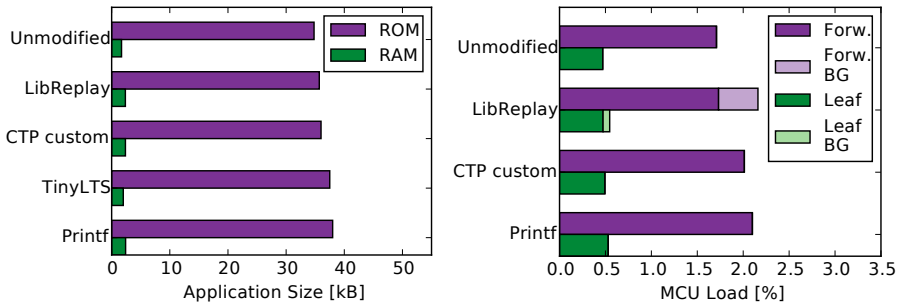
After introducing LibReplay and its architecture, we evaluate its performance. We begin with a set of micro benchmarks to determine MCU and memory efficiency. Next, we compare LibReplay to the state of art and show that its overhead is similar to today's approaches to logging while these commonly do not log sufficient information for providing replay capabilities. We implemented LibReplay in TinyOS 2.1.2 and evaluate using TelosB nodes.

4.1 MCU and Memory Efficiency of LibReplay

In LibReplay, logging consists of two steps: the fast logging itself to an in-memory buffer and a second low-priority background process that handles the heavy lifting to external storage. As a result, the logging itself has only minimal impact on the program execution (see Figure 3a). The RAM footprint of LibReplay strongly depends on the buffer size chosen (see Figure 3b). ROM is stable independent of the buffer size chosen. For the following, we use the default value of 300 bytes for the buffer. Our experience shows that this is sufficient for most application scenarios, and it is similar to the default setting in the state of the art. Nonetheless, when compared to the overall memory footprint of the application, the footprint of LibReplay stays small (see Figure 3c), leaving sufficient space for complex applications.

4.2 LibReplay and Traditional Approaches to Logging

We compare the efficiency of LibReplay to traditional logging approaches: printf, TinyLTS [11], and the customized logging layer of the Collection Tree Protocol



(a) The memory footprint of LibReplay is similar to traditional logging systems. The footprint of TinyLTS is taken from its publication [11], as the source code is not available to us.

(b) Average MCU load in a CTP network of 25 nodes. We distinguish leaf nodes and forwarders. For LibReplay we also disinguish between logging and the background (BG) process.

Figure 4. The memory footprint of LibReplay and its MCU load are similar to traditional logging approaches.The benchmark application is CTP routing (`TestNetwork`), we use the default buffer size of all loggers.

(CTP) [6]. Our results show that both the memory footprint and the MCU load of logging with LibReplay is comparable to these traditional approaches to logging (see Figure 4). We note that these, in contrast to LibReplay, commonly do not log sufficient information to enable replay debugging.

5 Conclusion

We introduced LibReplay, a lightweight architecture for flexible logging and deterministic replay in sensor networks. LibReplay enables (1) event logging with only a small intrusion of the system, and (2) deterministic event replay in controlled environments such as system simulators. As a result, we can exploit the debugging capabilities of modern system simulators. Overall, LibReplay simplifies bug hunting in deployed sensor networks and provides a debugging experience similar to debugging (local and non-distributed) sequential programs. We discuss the architecture of LibReplay and our performance evaluations show that the efficiency of LibReplay is similar to the state of the art, which commonly does not log sufficient information to provide replay capabilities. We have made the source code of LibReplay publicly available at `https://github.com/olafland/LibReplay`.

Acknowledgments

This work was partially supported by the EC, through project FP7-STREP-288195, KARYON (Kernel-based ARchitecture for safetY-critical cONtrol), and by the Swedish Energy Agency under the program Energy, IT and Design.

References

1. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: ATEC: Proc. of the Annual Conf. on USENIX Annual Technical Conference (2005)
2. Dong, W., et al.: Dynamic Logging with Dylog for Networked Embedded Systems. In: SECON: Proc. of the IEEE Int. Conf. on Sensing, Communication, and Networking (2014)
3. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN: Proc. of the IEEE Conf. on Local Computer Networks (2004)
4. Eriksson, J., et al.: Towards Interoperability Testing for Wireless Sensor Networks with COOJA/MSPSim. In: EWSN: Proc. of the European Conf. on Wireless Sensor Networks (2009)
5. Geels, D., et al.: Replay Debugging for Distributed Applications. In: ATEC: Proc. of the Annual Conf. on USENIX Annual Technical Conference (2006)
6. Gnawali, O., et al.: Collection Tree Protocol. In: SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems (2009)
7. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7) (1978)
8. Narayanasamy, S., et al.: BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In: ISCA: Proc. of the Annual Int. Symposium on Computer Architecture (2005)
9. Netzer, R.H.B., Miller, B.P.: Optimal Tracing and Replay for Debugging Message-passing Parallel Programs. In: Supercomputing: Proc. of the ACM/IEEE Conf. on Supercomputing (1992)
10. Österlind, F., et al.: Cross-Level Sensor Network Simulation with COOJA. In: LCN: Proc. of the IEEE Conf. on Local Computer Networks (2006)
11. Sauter, R., et al.: TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS. In: INFOCOM: Proc. of the IEEE Int. Conf. on Computer Communications (2011)
12. Sommer, P., Kusy, B.: Minerva: Distributed Tracing and Debugging in Wireless Sensor Networks. In: SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems (2013)
13. Sundaram, V., Eugster, P., Zhang, X.: Prius: Generic Hybrid Trace Compression for Wireless Sensor Networks. In: SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems (2012)
14. Sundaram, V., et al.: Diagnostic Tracing for Wireless Sensor Networks. ACM Trans. Sen. Netw. 9(4) (2013)
15. Tancreti, M., et al.: Aveksha: A Hardware-software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems. In: SenSys: Proc. of the ACM Conf. on Embedded Networked Sensor Systems (2011)
16. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable Sensor Network Simulation with Precise Timing. In: IPSN: Proc. of the ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (2005)
17. Wan, L., Cao, Q.: Towards Instruction Level Record and Replay of Sensor Network Applications. In: MASCOTS: Proc. of the IEEE Int. Symp. on Modeling, Analysis Simulation of Computer and Telecommunication Systems (2013)
18. Wang, M.o.: Dependence-based Multi-level Tracing and Replay for Wireless Sensor Networks Debugging. In: LCTES: Proc. of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (2011)